INTENTO — Official Manual 0.1

Index

Human-readable code. Real execution.

Project Introduction

INTENTO is a human-readable programming language designed around a simple idea: code should express intention clearly, while still being precise enough to execute.

The name is Italian. The syntax is English. The purpose is universal: to reduce the distance between what a human wants to do and what a computer can safely execute.

INTENTO is not free natural language. It is not a chatbot prompt. It is not a poetic layer over vague commands. INTENTO is a real programming language proposal, built on controlled human readability. Its instructions are meant to look natural, but they must follow deterministic rules. A line of INTENTO code should be understandable to a person and parseable by a machine.

This balance defines the project.

Many programming languages are powerful but intimidating. They require the programmer to think in symbols, structures, libraries, syntax rules, runtime behavior, and error models before they can express even simple intentions. INTENTO does not reject those technical realities. Instead, it tries to organize them in a more readable form.

For example, INTENTO prefers instructions such as:

```
remember title as "Human Readable Code"

show title
```

over compressed or cryptic syntax. The goal is not to hide computation. The goal is to make computation legible.

INTENTO also does not try to replace mature languages such as Python. Python can serve as a powerful backend layer for registered actions, libraries, file handling, data processing, automation, and system integration. INTENTO remains the intentional surface: the language in which the human expresses what should happen. The Runtime remains responsible for parsing, validation, safety, and execution.

This distinction is central. INTENTO source code should not contain raw Python or raw shell commands. Instead, advanced capabilities should be exposed through registered actions with clear names, declared inputs, declared outputs, and safety metadata.

A program should be able to say:

```
use action "text.slugify" with title and call the result slug
```

without exposing the implementation behind `text.slugify`. The backend may be Python. The INTENTO code remains readable, controlled, and safe.

The project has four guiding principles.

First, readability must be real. INTENTO should be understandable without forcing the reader to decode dense symbolic syntax.

Second, readability must be controlled. Human-readable does not mean ambiguous. Every command must have precise grammar and predictable behavior.

Third, execution must be safe. Operations that read files, write files, access the network, or interact with the system must be governed by permissions, workspace rules, and confirmation.

Fourth, power should be extensible. INTENTO should stay small at its core, while registered actions and the Standard Library allow it to grow into practical use.

This manual defines INTENTO as a language specification and execution model. It introduces the idea, the syntax, the Runtime, the Standard Library, modules, registered actions, safety levels, project structure, and implementation rules.

INTENTO begins from a human sentence-like surface, but it does not stop there.

Its ambition is not merely to be easy.

Its ambition is to be readable, executable, safe, and real.

Chapter 1 — The Idea Behind INTENTO

Why INTENTO exists, what problem it solves, and what “human-readable programming” means. This chapter defines INTENTO as a real programming language based on controlled human readability: natural on the surface, precise underneath.

Chapter 2 — Architecture and Execution Model

How INTENTO code becomes execution. This chapter explains `.intento` files, parsing, semantic checking, the safety engine, the Runtime, Python-backed execution, output, and logs.

Chapter 3 — Core Syntax: Instructions, Values, and Memory

The basic language core. This chapter explains program structure, instructions, comments, indentation, `show`, values, types, names, `remember`, and basic expressions with `plus`.

Chapter 4 — Interaction and Decisions

How programs receive input and choose what to do. This chapter explains `ask`, user answers, empty values, `if`, `else`, comparisons, validation, and `stop`.

Chapter 5 — Repetition, Lists, and Custom Actions

How programs avoid repetition and organize logic. This chapter explains `repeat`, lists, `for each`, custom actions with `to`, parameters, `return`, and reusable program parts.

Chapter 6 — Files, Safety, and Python-Backed Power

How INTENTO moves from simple scripts to real execution. This chapter explains files and folders, confirmation syntax, safe and blocked actions, Runtime safety rules, Python-backed operations, and future expansion.

Chapter 1 — The Idea Behind INTENTO

INTENTO is a programming language designed to make code closer to human intention.

Its goal is simple:

> write what you want the computer to do in a clear, readable form.

INTENTO is not free natural language.
It is not a chatbot.
It is not a poetic way to write commands.

INTENTO is a real programming language with a controlled syntax.

It uses sentences that look human, but every sentence must be precise enough for a computer to parse, check, and execute.

A basic INTENTO program looks like this:

show "Hello world"

This line is easy to read.

It means:

Display the text "Hello world".

The surface is simple.
The meaning is exact.

That is the central idea of INTENTO.

1.1 Human-readable, machine-parseable

INTENTO is built on one principle:

human-readable on the surface

machine-parseable underneath

A line of INTENTO should be easy for a person to understand.

At the same time, it must follow a strict pattern so the Runtime can interpret it without guessing.

For example:

remember name as "Aurora"

This line is readable for a human.

It also has a precise structure:

instruction: remember

name: name

value: "Aurora"

The Runtime can store the value `"Aurora"` under the name `name`.

This is different from a vague sentence like:

keep Aurora somewhere

That sentence may sound human, but it is not precise enough to be executed as code.

INTENTO chooses controlled readability over vague freedom.

1.2 Why INTENTO exists

Most programming languages ask humans to adapt to the machine.

They require symbols, strict forms, technical words, and concepts that may feel distant from normal thought.

This is necessary in many cases, but it also creates a barrier.

A person may know what they want:

I want to ask for a name.

I want to show a greeting.

I want to create a folder.

I want to read a file.

I want to repeat an action.

But they may not know the right syntax in Python, Bash, JavaScript, or another language.

INTENTO tries to reduce that distance.

It gives common programming ideas a more readable form:

ask "What is your name?" and call the answer name

show "Hello " plus name

The program asks a question, stores the answer, and shows a greeting.

The code still has rules, but the rules are close to normal human instruction.

1.3 Italian name, English syntax

The name **INTENTO** is Italian.

It means intention, purpose, direction, and aim.

The name was chosen because every program begins with an intention.

Before code, there is a human purpose.

The syntax is English because English is the common language of programming, documentation, tools, and open-source communities.

So the identity of the language is:

Italian name.

English syntax.

Human-readable code.

Real execution.

The name gives INTENTO its identity.

The English syntax makes it practical and international.

1.4 Simple does not mean weak

INTENTO should be simple to read, but it must not be a toy language.

A simple surface can still support serious execution.

This program is simple:

```
remember price as 10

remember tax as 2

remember total as price plus tax


show total
```

But it already uses important programming concepts:

memory

names

numbers

expressions

output

INTENTO begins with a small core, but it is designed to grow.

Future versions may support modules, data tools, Python-backed capabilities, terminal translation, and compilation paths.

The goal is not to make programming childish.

The goal is to make programming clearer.

1.5 INTENTO and the computer

A computer cannot execute vague wishes.

It needs instructions.

INTENTO turns readable intentions into structured instructions.

For example:

with confirmation:

```
create folder "Documents/Aurora"
```

This is easy to read.

It also contains enough structure for execution:

confirmation block

action: create folder

path: Documents/Aurora

A Runtime can translate this into a safe file-system operation, such as a Python call or a terminal-level command.

The human writes the intention.

The Runtime handles the technical execution.

1.6 The role of safety

INTENTO is designed for real actions, so safety is part of the language.

Some instructions only display information:

show "Welcome"

These are safe.

Other instructions change something:

create folder "Documents/Test"

These may require confirmation.

Some actions are too dangerous for the first version of the language:

delete folder "Documents"

In INTENTO 0.1, destructive operations should be blocked or heavily restricted.

The language should help humans control the machine, not lose control of it.

1.7 The role of AI

AI can help users write INTENTO code.

A user may say:

Create a folder for my project.

An AI assistant may suggest:

with confirmation:

```
create folder "Documents/MyProject"
```

But AI is not the Runtime.

AI may suggest code.
INTENTO defines what the code means.
The Runtime executes according to fixed rules.

This keeps the language deterministic.

The same code must keep the same meaning every time it runs.

1.8 The core promise

The core promise of INTENTO is:

simple syntax

clear meaning

controlled execution

human understanding

INTENTO should let people write programs that are easy to read and precise enough to run.

It should not hide what happens.

It should not execute unclear instructions.

It should make the path from intention to action visible.

In one sentence:

> INTENTO is a human-readable programming language for expressing clear intentions that a computer can safely understand and execute.

Chapter 2 — Architecture and Execution Model

INTENTO code is written for humans, but it must be executed by a computer.

For this reason, INTENTO has two sides:

readable surface

technical execution model

The readable surface is what the user writes:

show "Hello world"

The execution model is what the Runtime does internally to understand and run that instruction.

INTENTO 0.1 is designed to run through an interpreter.

The interpreter reads a `.intento` file, checks it, builds an internal structure, applies safety rules, and executes the program through a backend.

The first backend is Python.

2.1 The execution chain

The basic execution chain is:

.intento file

→ source reader

→ lexer

→ parser

→ semantic checker

→ safety engine

→ runtime executor

→ backend

→ result

→ execution log

Each stage has a clear task.

The source reader loads the program.

The lexer splits the program into meaningful pieces.

The parser checks the grammar and builds structured instructions.

The semantic checker verifies that the instructions make sense.

The safety engine checks whether an action is safe, needs confirmation, or must be blocked.

The runtime executor runs the program.

The backend performs real operations through Python, the file system, or controlled terminal-level actions.

The result is shown to the user.

The execution log records what happened.

2.2 Source files

An INTENTO program is stored in a file with the `.intento` extension.

Example:

hello.intento

Content:

show "Hello from INTENTO"

A program can be executed with a command such as:

python3 intento.py hello.intento

This command starts the INTENTO interpreter and gives it the source file to run.

The exact command may change in future versions, but the principle remains the same:

the Runtime receives a .intento file and executes it according to INTENTO rules

---

2.3 Source reader

The source reader opens the `.intento` file and reads its content.

At this stage, the Runtime should detect basic file problems.

Example errors:

File not found: hello.intento

Cannot read file: permission denied

Unsupported file encoding

Empty program

The source reader does not understand the language yet.

It only loads the text and prepares it for analysis.

2.4 Lexer

The lexer turns source text into tokens.

A token is a small meaningful unit of code.

For example:

remember name as "Aurora"

can become:

KEYWORD remember

IDENTIFIER name

KEYWORD as

TEXT "Aurora"

Another example:

show "Hello"

can become:

KEYWORD show

TEXT "Hello"

The lexer recognizes things such as:

keywords

names

text values

numbers

booleans

list brackets

indentation

new lines

comments

The lexer does not execute anything.

It only prepares the program for the parser.

2.5 Parser

The parser checks whether tokens form valid INTENTO instructions.

This is valid:

show "Hello"

This is not valid:

show

The second example is incomplete because `show` needs a value to display.

A good parser error should be clear:

Syntax error on line 1: `show` needs a value.

The parser also checks blocks.

Example:

```
if name is empty:
    show "Name is required"
```

The indented line belongs to the `if` block.

If indentation is wrong, the parser should report it clearly.

2.6 Internal representation

After parsing, the program becomes an internal structure.

This structure can be called the **Intent Tree**.

Example source code:

remember name as "Aurora"

show "Hello " plus name

Possible internal structure:

Program

├── Remember name = "Aurora"

└── Show ("Hello " plus name)

The Intent Tree is useful because the Runtime does not need to guess from raw text.

It can work with structured instructions.

This makes execution more reliable.

2.7 Semantic checker

A program can have correct syntax but still be wrong.

Example:

show username

This line has valid structure, but it is a problem if `username` was never created with `remember`, `ask`, or another instruction.

The semantic checker should detect this before execution.

Possible error:

Semantic error on line 1: unknown name `username`.

The semantic checker may detect:

unknown names

invalid value types

invalid comparisons

wrong action parameters

invalid return usage

unsupported operations

The goal is simple: catch mistakes before they become runtime failures.

2.8 Safety engine

The safety engine checks instructions that may affect the outside world.

This includes file operations, folder operations, terminal-level actions, and future Python-backed capabilities.

Example:

show "Hello"

Category:

safe

Example:

create folder "Documents/Test"

Category:

needs confirmation

Example:

delete folder "Documents"

Category in INTENTO 0.1:

blocked

The safety engine should work on structured instructions, not raw shell strings.

A structured instruction like:

CreateFolder(path = "Documents/Test")

is easier and safer to inspect than arbitrary Bash code.

2.9 Runtime executor

The runtime executor runs the checked program.

It manages:

memory

input

output

conditions

loops

custom actions

errors

confirmations

logs

Example:

remember name as "Aurora"

show "Hello " plus name

Runtime behavior:

1. Store name = "Aurora".

2. Read name.

3. Join "Hello " and "Aurora".

4. Display "Hello Aurora".

The runtime executor must be deterministic.

It executes what the program says, not what it imagines the user might have meant.

2.10 Python backend

The first INTENTO backend should be Python.

Python is useful because it is stable, widely available, readable, and powerful.

Example:

show "Hello"

can be executed internally as:

print("Hello")

Example:

create folder "Documents/Test"

can be executed internally with Python file-system tools:

Path.home().joinpath("Documents/Test").mkdir(parents=True, exist_ok=True)

This does not mean that the user writes Python.

The user writes INTENTO.

Python provides execution power underneath.

This gives INTENTO a simple surface and a serious backend.

2.11 Terminal translation

Some INTENTO actions can also correspond to terminal-level operations.

Example:

create folder "Documents/Test"

Terminal equivalent:

mkdir -p ~/Documents/Test

However, INTENTO 0.1 should not blindly generate and run shell commands.

Whenever possible, the Runtime should prefer controlled backend operations.

Terminal translation is useful for documentation, explanation, debugging, and future Human Terminal OS integration.

A safe Runtime may show:

INTENTO action:

create folder "Documents/Test"

Possible terminal equivalent:

mkdir -p ~/Documents/Test

This helps the user learn what happens underneath without forcing them to write Bash directly.

2.12 Result

At the end of execution, the Runtime should return a clear result.

Successful example:

Program finished successfully.

Output example:

Hello Aurora

Blocked action example:

Program stopped.

Action blocked: destructive folder deletion is not allowed in INTENTO 0.1.

Error example:

Runtime error on line 4: cannot read file "Documents/missing.txt".

Runtime messages should be short, specific, and understandable.

A useful error tells the user:

what went wrong

where it happened

how to fix it when possible

2.13 Execution log

The execution log records what happened during a program run.

It is different from normal output.

Output is what the program shows to the user.

The log is what the Runtime records for review.

A basic log may include:

timestamp

program file

runtime version

instructions executed

confirmed actions

blocked actions

errors

final status

Example:

time: 2026-06-13T10:30:00

program: project.intento

runtime: intento-runtime 0.1

action: create folder Documents/Aurora

safety: needs confirmation

confirmed: yes

result: success

Logs are especially important when INTENTO works with files, folders, terminal translation, or Human Terminal OS.

2.14 Debug mode

A future Runtime should support debug mode.

Example command:

python3 intento.py program.intento --debug

Debug mode may show internal stages:

Reading source...

Lexing...

Parsing...

Checking semantics...

Checking safety...

Executing...

Writing log...

Debug mode is for developers, language designers, and advanced users.

Normal users should see clean output.

2.15 Summary

INTENTO is simple to read, but it needs a real execution model.

The Runtime does not execute vague text.

It follows a clear chain:

source

→ tokens

→ parsed instructions

→ semantic checks

→ safety checks

→ runtime execution

→ backend operation

→ result

→ log

This architecture keeps INTENTO readable for humans and precise enough for computers.

The surface language stays simple.

The internal system stays disciplined.

Chapter 3 — Core Syntax: Instructions, Values, and Memory

This chapter introduces the practical core of INTENTO.

After the idea of the language and its execution model, the reader now needs to understand how an INTENTO program is actually written. The goal is not to memorize many rules at once. The goal is to see the basic shape of the language: instructions, values, memory, output, and simple expressions.

INTENTO code should be easy to read, but it must still be exact. The Runtime cannot guess the programmer’s intention. It can only execute instructions that follow clear rules.

A minimal INTENTO program looks like this:

show "Hello from INTENTO"

Output:

Hello from INTENTO

This one line is a complete program. It contains one instruction, `show`, and one value, `"Hello from INTENTO"`.

3.1 Source files

An INTENTO program is written in a plain text file with the `.intento` extension.

Example:

hello.intento

The extension tells the Runtime that the file contains INTENTO code. The file itself is still a normal text file and can be edited with any text editor.

A small file named `hello.intento` may contain:

show "Hello world"

Output:

Hello world

In early versions, the file may be executed through the Runtime with a command such as:

python3 intento.py hello.intento

Expected terminal output:

Hello world

In future versions, this may become simpler:

intento run hello.intento

The execution command may evolve, but the source rule remains the same: INTENTO programs live in `.intento` files, and the Runtime reads those files from top to bottom.

3.2 Instructions

An instruction is a command written in INTENTO syntax.

The basic rule is simple: one instruction per line.

Example:

show "Starting"

show "Working"

show "Finished"

Output:

Starting

Working

Finished

The Runtime executes the first line, then the second, then the third. Unless a condition, repetition, action call, or `stop` changes the flow, execution is top-to-bottom.

An INTENTO instruction must begin with a known keyword or a known custom action. For example, `show` is a keyword. It tells the Runtime to display a value.

This is valid:

show "Program started"

Output:

Program started

This is not valid:

display "Program started"

Expected error:

Syntax error: unknown instruction `display`.

The error is correct because `display` is not part of INTENTO Core 0.1. The official output instruction is `show`.

INTENTO avoids multiple instructions on the same line. This keeps the code readable and keeps parsing simple. A program should not be written like this:

show "Starting" show "Finished"

Expected error:

Syntax error: unexpected instruction after value.

The correct version is:

show "Starting"

show "Finished"

Output:

Starting

Finished

3.3 Blank lines and comments

Blank lines are allowed. They do not change the program result.

Example:

show "First"

show "Second"

Output:

First

Second

The blank line only improves readability.

Comments begin with #. A comment is a note for humans. The Runtime ignores it.

Example:

This program prints a greeting

show "Hello"

Output:

Hello

Comments should explain intention, context, or reason. They should not repeat obvious code. This is useful:

The name is used later in the greeting

remember name as "Aurora"

show "Hello " plus name

Output:

Hello Aurora

The comment explains why the value is remembered. That is better than a comment such as `# show hello`, which adds no real information.

3.4 Blocks and indentation

Some instructions control other instructions. These controlled instructions form a block.

A block starts after a line ending with `:`. The instructions inside the block are indented.

Example:

remember name as empty

if name is empty:

    show "Name is required"

Output:

Name is required

The line `if name is empty:` starts a block. The indented line belongs to that block. It runs only if the condition is true.

Indentation is not decoration. It is part of the syntax. INTENTO Core uses four spaces as the recommended indentation.

This is invalid:

remember name as empty

if name is empty:

show "Name is required"

Expected error:

Syntax error: expected an indented block after `if`.

Blocks are used by conditions, alternatives, repetition, list loops, confirmation sections, and custom actions. The exact commands are introduced across the next chapters, but the structure is always the same: a block starter ends with `:`, and the controlled instructions are indented below it.

A block can also contain another block:

```
remember name as empty


if name is empty:
    show "Checking name..."
    if name is empty:
        show "Name is still missing"
```

Output:

```
Checking name...
Name is still missing
```

Nested blocks are valid, but they should remain readable. If the indentation becomes too deep, the program should usually be reorganized with custom actions.

3.5 Values and types

A value is a piece of information used by the program.

INTENTO Core 0.1 uses these basic value kinds: text, number, boolean, empty, and list.

Text is written between quotation marks:

```
show "INTENTO"
```

Output:

```
INTENTO
```

Numbers are written without quotation marks:

```
show 42
```

Output:

```
42
```

Booleans are written as `true` or `false`:

show true

show false

Output:

true

false

The value `empty` represents absence of content:

show empty

Output:

(empty line)

A list contains multiple values in order:

show ["Linux", "Python", "INTENTO"]

Output:

["Linux", "Python", "INTENTO"]

Types matter because the Runtime must know how to handle each value. The text `"42"` and the number `42` may look similar when displayed, but they are not the same value inside the program.

Example:

show "42" plus "8"

show 42 plus 8

Output:

428

50

The first expression joins text. The second expression adds numbers. INTENTO does not silently guess that text should become a number. If conversion is needed, it must be explicit.

3.6 Output with `show`

The `show` instruction displays a value.

Its basic form is:

show value

The value can be text, a number, a boolean, an empty value, a list, a remembered name, or an expression.

Example:

show "Welcome"

show 100

show true

show ["one", "two"]

Output:

Welcome

100

true

["one", "two"]

Text must be quoted. Without quotation marks, the Runtime treats words as names or keywords.

This is invalid:

show Welcome

Expected error:

Semantic error: unknown name `Welcome`.

The correct version is:

show "Welcome"

Output:

Welcome

This distinction is important because a remembered name is written without quotation marks:

remember name as "Aurora"

show name

show "name"

Output:

Aurora

name

The first `show` displays the value stored in `name`. The second `show` displays the literal text `"name"`.

The `show` instruction does not change memory, does not create files, and does not ask for input. It only sends a value to the output. For this reason, it is considered a safe instruction.

3.7 Memory with `remember`

Programs need memory because they often need to use a value more than once.

INTENTO stores values with `remember`.

The basic form is:

remember name as value

Example:

remember language as "INTENTO"

show language

Output:

INTENTO

The instruction `remember language as "INTENTO"` stores the text value `"INTENTO"` under the name `language`. Later, `show language` retrieves that value and displays it.

The `remember` instruction does not display anything by itself:

remember language as "INTENTO"

Output:

(no visible output)

The value exists in program memory, but nothing is shown until a `show` instruction uses it.

A remembered value must exist before it can be used:

show project

Expected error:

Semantic error: unknown name `project`.

Correct version:

remember project as "INTENTO"

show project

Output:

INTENTO

A name should be clear and easy to read. In INTENTO Core 0.1, a name should start with a lowercase letter and may contain lowercase letters, numbers, and underscores. It should not contain spaces and should not be a keyword.

Valid names include `name`, `project_name`, `total_price`, and `item2`.

This is invalid:

remember project name as "INTENTO"

Expected error:

Syntax error: names cannot contain spaces.

The correct version is:

remember project_name as "INTENTO"

show project_name

Output:

INTENTO

A remembered value can also be replaced by remembering a new value with the same name:

remember status as "waiting"

show status

remember status as "running"

show status

Output:

waiting

running

This is useful when a program state changes during execution.

3.8 Expressions with `plus`

An expression is code that produces a value.

The simplest expression is a value itself:

```
show "Hello"
```

Output:

```
Hello
```

A remembered name is also an expression:

```
remember name as "Aurora"

show name
```

Output:

```
Aurora
```

INTENTO Core 0.1 uses `plus` as the main basic expression operator.

When both values are numbers, `plus` performs numeric addition:

```
show 10 plus 5
```

Output:

```
15
```

When text is involved, `plus` joins values into text:

```
remember name as "Aurora"

show "Hello " plus name
```

Output:

```
Hello Aurora
```

Spaces are not inserted automatically. If the programmer wants a space, the space must be written inside the text.

Example:

```
remember first_name as "Ada"

remember last_name as "Lovelace"


show first_name plus last_name

show first_name plus " " plus last_name
```

Output:

```
AdaLovelace

Ada Lovelace
```

The result of an expression can be stored with `remember`:

```
remember price as 30

remember tax as 5

remember total as price plus tax


show "Total: " plus total
```

Output:

```
Total: 35
```

This is the recommended style when a calculation has meaning. First calculate and name the value. Then show it.

The official INTENTO Core 0.1 form is `plus`. A future Runtime may allow `+` as an optional shorthand, but this manual uses `plus` because it preserves the human-readable identity of the language.

Invalid type combinations should produce clear errors:

```
show true plus false
```

Expected error:

```
Type error: cannot use `plus` with boolean and boolean.
```

The Runtime should not guess a meaning where the language has not defined one.

3.9 A complete core example

This example uses the core elements introduced in this chapter: comments, remembered values, types, expressions, output, and a simple block.

```
# Basic user profile

remember username as "alessandro"
remember display_name as "Alessandro"
remember age as 42
remember active as true
remember notes as empty
remember skills as ["Linux", "Python", "INTENTO"]

show "User profile"
show "Username: " plus username
show "Display name: " plus display_name
show "Age: " plus age
show "Active: " plus active
show "Skills: " plus skills

if notes is empty:
    show "Notes: no notes available"
```

Output:

```
User profile
Username: alessandro
Display name: Alessandro
Age: 42
Active: true
```

Skills: ["Linux", "Python", "INTENTO"]

Notes: no notes available

This program is still small, but it already shows the core logic of INTENTO. Values are remembered with clear names. Output is produced with `show`. Text, numbers, booleans, empty values, and lists are handled predictably. The condition at the end shows how a block can control whether an instruction runs.

The program is readable for a human, but it is also structured enough for the Runtime to parse and execute.

3.10 Summary

The core syntax of INTENTO is based on a few simple rules.

A program is written in a `.intento` file. It contains one instruction per line. Blank lines are allowed. Comments start with #. Blocks start with `:` and use indentation.

The `show` instruction displays values. The `remember` instruction stores values in memory. Values have types, such as text, number, boolean, empty, and list. Names refer to remembered values. Expressions produce new values, and `plus` is the basic operator for numeric addition and text joining.

The important principle is this: INTENTO should be easy to read, but never vague. Every instruction must have a clear meaning for the human reader and a precise meaning for the Runtime.

Chapter 4 — Interaction and Decisions

A program becomes more useful when it can interact with the user and choose what to do.

The previous chapter introduced the core elements of INTENTO: instructions, values, memory, output, and simple expressions. Those elements are enough to write fixed programs, but real programs often need something more. They need to ask questions, receive answers, check values, handle missing information, and follow different paths.

This chapter introduces the instructions and structures that allow INTENTO programs to interact and decide: `ask`, `if`, `else`, comparisons, and `stop`.

The goal is not to make programs complicated. The goal is to make them responsive. A program should not blindly continue when information is missing or invalid. It should check what it knows, explain what is wrong, and continue only when continuing makes sense.

4.1 Asking for input with `ask`

The `ask` instruction receives information from the user while the program is running.

Its basic form is:

```
ask "Question?" and call the answer name
```

The question must be written as text, between quotation marks. The name after `and call the answer` is the memory name that will store the user’s answer.

Example:

```
ask "What is your name?" and call the answer name

show "Hello " plus name
```

Example interaction:

```
What is your name?

Alessandro

Hello Alessandro
```

The Runtime first displays the question. Then it waits for the user to type an answer. When the user writes `Alessandro`, the Runtime stores that answer under the name `name`. The following instruction can then use `name` like any other remembered value.

The `ask` instruction is similar to `remember`, but the source of the value is different. With `remember`, the value is written directly in the program. With `ask`, the value comes from the user during execution.

This program uses `remember`:

```
remember name as "Aurora"

show "Hello " plus name
```

Output:

```
Hello Aurora
```

This program uses `ask`:

```
ask "What is your name?" and call the answer name

show "Hello " plus name
```

Example interaction:

```
What is your name?

Aurora

Hello Aurora
```

Both programs create a value called `name`. The difference is that one value is fixed in the source code, while the other is provided at runtime.

4.2 Input is usually text

In INTENTO Core 0.1, the answer received by `ask` should be treated as text unless the Runtime provides an explicit conversion feature.

This matters because the user may type characters that look like a number, but the Runtime still receives them as an answer.

Example:

ask "Enter your age:" and call the answer age

show "You typed: " plus age

Example interaction:

Enter your age:

42

You typed: 42

The visible output is `42`, but the value stored in `age` should be understood as user input, normally text. If a program needs to perform numeric calculations with that answer, the conversion should be explicit in a later language feature or through a Python-backed operation.

INTENTO should not silently guess that every input that looks numeric is automatically a number. Silent guessing can create errors. A user might type `042`, `42 years`, or nothing at all. The program should handle those cases clearly.

In this chapter, `ask` is mainly used for text input and validation, especially checking whether an answer is empty.

4.3 Empty answers

A user may answer a question with nothing.

For example, the program may ask for a name, and the user may simply press Enter. In that case, the answer is empty.

The value `empty` represents absence of content.

Example:

```
ask "What is your name?" and call the answer name

if name is empty:
    show "Name is required"
```

Example interaction with an empty answer:

```
What is your name?

Name is required
```

The blank line after the question represents the user pressing Enter without typing a value. The condition `name is empty` becomes true, so the message is displayed.

If the user provides a value, the block does not run:

```
ask "What is your name?" and call the answer name

if name is empty:
    show "Name is required"

show "Program finished"
```

Example interaction:

```
What is your name?

Alessandro

Program finished
```

The program does not show `Name is required` because the answer is not empty.

This is one of the most common uses of conditions: checking whether required information exists before continuing.

4.4 Decisions with `if`

The `if` structure allows a program to run instructions only when a condition is true.

Its basic form is:

```
if condition:
    instruction
```

The line with `if` ends with `:`. The instructions controlled by the condition are indented below it.

Example:

```
remember name as empty


if name is empty:
    show "Name is required"
```

Output:

```
Name is required
```

The condition is true because `name` contains `empty`. Therefore, the indented instruction runs.

Now compare this version:

```
remember name as "Aurora"


if name is empty:
    show "Name is required"


show "End"
```

Output:

```
End
```

Here the condition is false, so the indented `show` instruction is skipped. The final instruction is outside the block, so it still runs.

This distinction is important. Indentation tells the Runtime which instructions belong to the `if` block and which instructions continue normally after the block.

Invalid indentation should produce a clear error:

```
remember name as empty
```

if name is empty:

show "Name is required"

Expected error:

Syntax error: expected an indented block after `if`.

The condition controls only the indented block. Without indentation, the structure is not valid.

4.5 Alternatives with `else`

The `else` structure defines what should happen when an `if` condition is false.

Example:

remember name as "Aurora"

```
if name is empty:
    show "Name is required"
else:
    show "Hello " plus name
```

Output:

Hello Aurora

The condition `name is empty` is false, so the `if` block is skipped and the `else` block runs.

With an empty name, the result changes:

remember name as empty

```
if name is empty:
    show "Name is required"
else:
    show "Hello " plus name
```

Output:

Name is required

The `else` block must belong to an `if`. It cannot appear alone.

Invalid:

```
else:
    show "This has no condition"
```

Expected error:

Syntax error: `else` must follow an `if` block.

The purpose of `else` is to keep decisions readable. Instead of writing two separate conditions, the programmer can express two paths clearly: if this is true, do one thing; otherwise, do another.

4.6 Comparisons

A condition usually contains a comparison.

A comparison checks a relationship between values and produces a boolean result: true or false.

The most basic comparison is `is`.

Example:

```
remember language as "INTENTO"


if language is "INTENTO":
    show "Correct language"
else:
    show "Different language"
```

Output:

Correct language

The comparison `language is "INTENTO"` checks whether the value stored in `language` is equal to the text `"INTENTO"`.

The opposite comparison is `is not`.

Example:

remember language as "Python"

if language is not "INTENTO":

    show "This is not INTENTO"

else:

    show "This is INTENTO"

Output:

This is not INTENTO

For numbers, INTENTO Core may support readable numeric comparisons such as `is greater than`, `is less than`, `is at least`, and `is at most`.

Example:

remember age as 20

if age is at least 18:

    show "Access allowed"

else:

    show "Access denied"

Output:

Access allowed

Another example:

remember temperature as 15

if temperature is less than 18:

    show "Room is cold"

else:

    show "Room is warm enough"

Output:

Room is cold

Comparisons should be type-aware. A Runtime should not compare unrelated types by guessing.

Invalid:

remember age as "20"

if age is at least 18:

    show "Access allowed"

else:

    show "Access denied"

Expected error:

Type error: cannot compare text with number using `is at least`.

The value `"20"` is text because it is written between quotation marks. The value `18` is a number. If the program needs numeric comparison, `age` must contain a number or be explicitly converted before comparison.

This strictness protects the meaning of the program.

4.7 Booleans in decisions

A boolean value can be used directly in a decision.

Example:

remember active as true

if active is true:

    show "System is active"

else:

    show "System is not active"

Output:

System is active

With `false`, the other path runs:

remember active as false

if active is true:

    show "System is active"

else:

    show "System is not active"

Output:

System is not active

The values `true` and `false` are booleans, not text. This is valid:

remember active as true

This stores text, not a boolean:

remember active as "true"

A strict Runtime should not treat them as the same value.

Invalid:

remember active as "true"

if active is true:

    show "System is active"

else:

    show "System is not active"

Expected error:

Type error: cannot compare text with boolean.

Correct version:

remember active as true

```
if active is true:
    show "System is active"
else:
    show "System is not active"
```

Output:

```
System is active
```

This rule keeps conditions precise. Text that looks like a boolean is still text.

4.8 Stopping execution with `stop`

The `stop` instruction ends the program immediately.

It is useful when continuing would not make sense, especially after invalid or missing input.

Example:

```
ask "Project name?" and call the answer project

if project is empty:
    show "Project name is required"
    stop

show "Creating project: " plus project
```

Example interaction with an empty answer:

```
Project name?

Project name is required
```

The final `show` instruction does not run because `stop` ends the program.

With a valid answer, the program continues:

```
ask "Project name?" and call the answer project


if project is empty:
    show "Project name is required"
    stop


show "Creating project: " plus project
```

Example interaction:

```
Project name?
INTENTO
Creating project: INTENTO
```

The condition is false, so the block is skipped. The program reaches the final instruction and displays the project name.

The `stop` instruction does not need a value.

Invalid:

```
stop "finished"
```

Expected error:

```
Syntax error: `stop` does not take a value.
```

If the programmer wants to show a message before stopping, the message should be written as a separate instruction:

```
show "Program finished"
stop
```

Output:

```
Program finished
```

After `stop`, no following instruction is executed.

4.9 Designing clear validation

Validation means checking whether a value is acceptable before using it.

In INTENTO, validation should be written clearly. A program should explain what went wrong and stop or choose another path when necessary.

Example:

```
ask "Project name?" and call the answer project


if project is empty:
    show "Project name is required"
    stop


show "Project accepted: " plus project
```

Example interaction with a valid value:

```
Project name?
Aurora
Project accepted: Aurora
```

Example interaction with an empty value:

```
Project name?


Project name is required
```

This program is small, but it follows a good pattern. It asks for a value, checks the value, reports a clear problem if the value is missing, and continues only when the value is usable.

A less helpful program would be:

```
ask "Project name?" and call the answer project
show "Creating project: " plus project
```

Example interaction with an empty value:

```
Project name?
```

Creating project:

The program does not crash, but it produces unclear behavior. It tries to continue even though the project name is missing.

Good INTENTO programs should avoid this. They should not only execute; they should communicate.

4.10 A complete interaction example

This example combines input, memory, empty checking, `if`, `else`, comparisons, and `stop`.

```
show "Account check"


ask "What is your name?" and call the answer name


if name is empty:
    show "Name is required"
    stop


remember minimum_age as 18
remember user_age as 20


show "Hello " plus name


if user_age is at least minimum_age:
    show "Access allowed"
else:
    show "Access denied"


show "Check complete"
```

Example interaction:

Account check

What is your name?

Alessandro

Hello Alessandro

Access allowed

Check complete

If the user does not enter a name, the output is different:

Account check

What is your name?

Name is required

The program stops after the missing name message. It does not continue to the access check.

This example intentionally uses `remember user_age as 20` instead of asking for age. In INTENTO Core 0.1, user input should be treated as text unless explicit conversion is available. Numeric input conversion belongs to a later feature or to Python-backed execution. This keeps the example precise and prevents the Runtime from guessing.

The important structure is clear: ask for required information, validate it, stop if it is missing, and continue only when the program has what it needs.

4.11 Common errors

A common error is writing `ask` without a quoted question.

Invalid:

ask What is your name? and call the answer name

Expected error:

Syntax error: question text must be written between quotation marks.

Correct:

ask "What is your name?" and call the answer name

show name

Example interaction:

What is your name?

Aurora

Aurora

Another common error is forgetting the name that should store the answer.

Invalid:

ask "What is your name?"

Expected error:

Syntax error: `ask` must call the answer by name.

Correct:

ask "What is your name?" and call the answer name

show "Hello " plus name

Example interaction:

What is your name?

Aurora

Hello Aurora

Another common error is writing an `if` condition without a colon.

Invalid:

remember name as empty

if name is empty

    show "Name is required"

Expected error:

Syntax error: expected `:` after condition.

Correct:

```
remember name as empty


if name is empty:
    show "Name is required"
```

Output:

```
Name is required
```

These errors are not only technical details. They protect the structure of the language. The Runtime needs clear boundaries: where the question starts and ends, where the answer is stored, where a block begins, and which instructions belong to that block.

4.12 Summary

Interaction begins with `ask`. It lets the program receive information from the user and store that answer under a name.

Decisions begin with `if`. They let the program run a block only when a condition is true. The `else` block defines the alternative path when the condition is false.

Comparisons such as `is`, `is not`, `is greater than`, `is less than`, `is at least`, and `is at most` allow the Runtime to check values. Comparisons should be type-aware and should not depend on hidden guessing.

The `stop` instruction ends the program immediately. It is especially useful after validation errors, when continuing would produce unclear or unsafe behavior.

The central idea of this chapter is simple: an INTENTO program should not only execute instructions. It should react to values, explain problems, and choose clear paths.

Readable code is not enough if behavior is unclear. INTENTO aims for both: readable structure and predictable execution.

Chapter 5 — Repetition, Lists, and Custom Actions

A program should not repeat the same code by hand when the same idea can be expressed once and reused.

The previous chapters introduced fixed instructions, memory, output, input, and decisions. Those tools are enough for small programs, but larger programs need structure. They need to repeat actions, process collections of values, and define reusable pieces of behavior.

This chapter introduces three important parts of INTENTO: repetition with `repeat`, lists with `for each`, and custom actions with `to`.

The goal is not to make the language more complex. The goal is to keep programs readable when they grow.

5.1 Repetition with `repeat`

The `repeat` instruction runs a block more than once.

Its basic form is:

```
repeat number times:
    instruction
```

The line with `repeat` ends with `:`. The indented instructions below it form the block that will be repeated.

Example:

```
repeat 3 times:
    show "Hello"
```

Output:

```
Hello
Hello
Hello
```

The Runtime reads `repeat 3 times:` and executes the indented block three times. The instruction `show "Hello"` is written only once, but it runs three times.

A repeated block may contain more than one instruction.

```
repeat 2 times:
    show "Preparing"
    show "Done"
```

Output:

```
Preparing
Done
Preparing
```

Done

The whole block is repeated, not only the first line. This is why indentation is important: it tells the Runtime which instructions belong to the repetition.

The repeat count must be a number. In INTENTO Core 0.1, it should be a whole number greater than or equal to zero.

Invalid:

```
repeat "3" times:
    show "Hello"
```

Expected error:

Type error: repeat count must be a number.

The value `"3"` is text, not a number. INTENTO should not guess that the programmer meant the number `3`.

A remembered number can also be used as the repeat count:

```
remember attempts as 3


repeat attempts times:
    show "Trying..."
```

Output:

Trying...

Trying...

Trying...

This is useful when the number of repetitions has a meaning in the program.

5.2 Repetition and memory

A repeated block can use and change remembered values.

Example:

```
remember counter as 0
```

```
repeat 3 times:
    remember counter as counter plus 1
    show counter
```

Output:

```
1
2
3
```

The program begins with `counter` equal to `0`. Each repetition calculates `counter plus 1`, stores the result back into `counter`, and shows the new value.

This is a common pattern: initialize a value, repeat a block, and update the value inside the block.

The order matters. This version gives a different output:

```
remember counter as 0


repeat 3 times:
    show counter
    remember counter as counter plus 1
```

Output:

```
0
1
2
```

Both programs are valid. They simply express different intentions. In the first program, the counter is increased before it is shown. In the second program, the counter is shown before it is increased.

INTENTO does exactly what the instructions say, in order.

5.3 Lists

A list is a value that contains multiple values in order.

A list is written between square brackets.

Example:

remember names as ["Ada", "Grace", "Linus"]

show names

Output:

["Ada", "Grace", "Linus"]

The value `names` is one list containing three text values. The order is part of the list: `"Ada"` is first, `"Grace"` is second, and `"Linus"` is third.

A list can contain numbers:

remember numbers as [1, 2, 3]

show numbers

Output:

[1, 2, 3]

A list can also be empty:

remember items as []

show items

Output:

[]

An empty list is not the same as `empty`. The value `empty` means absence of content. The value `[]` means that a list exists, but it contains no items.

This difference matters because a program can still process an empty list safely. It simply has no elements to process.

5.4 Processing lists with `for each`

The `for each` instruction runs a block once for every item in a list.

Its basic form is:

for each item in list:

    instruction

The word after `each` is the temporary name used for the current item. The name after `in` must refer to a list.

Example:

```
remember names as ["Ada", "Grace", "Linus"]


for each name in names:
    show name
```

Output:

```
Ada
Grace
Linus
```

The Runtime takes the first item in `names`, stores it temporarily as `name`, and runs the block. Then it takes the second item, stores it as `name`, and runs the block again. It continues until the list is finished.

The temporary name is used inside the block.

Example:

```
remember names as ["Ada", "Grace", "Linus"]


for each name in names:
    show "Member: " plus name
```

Output:

```
Member: Ada
Member: Grace
Member: Linus
```

The list itself does not change. The loop only reads each item and executes the block.

If the list is empty, the block does not run.

```
remember names as []


for each name in names:
    show name
```

show "Finished"

Output:

Finished

This is not an error. It simply means there are no items to process.

The value after `in` must be a list.

Invalid:

remember name as "Ada"

for each letter in name:

    show letter

Expected error:

Type error: `for each` requires a list after `in`.

A future version of INTENTO may support iterating through text characters, but Core 0.1 should keep the rule simple: `for each` works with lists.

5.5 Custom actions with `to`

A custom action is a reusable block of instructions.

It is defined with `to`.

The basic form is:

to action_name:

    instruction

Defining an action does not run it. It only teaches the Runtime that the action exists.

Example:

to greet:

    show "Hello"

show "Ready"

Output:

Ready

The action `greet` is defined, but it is never called. Therefore, it does not display `"Hello"`.

To run the action, the program must call it by name:

```
to greet:
    show "Hello"


greet
```

Output:

Hello

Custom actions are useful because they give a name to a repeated intention. Instead of copying the same instructions in many places, the programmer can define the action once and call it when needed.

Example:

```
to show_separator:
    show "----------"


show "Start"
show_separator
show "End"
```

Output:

Start

End

The action name should be clear. In INTENTO Core 0.1, action names should follow the same practical style as memory names: lowercase words with underscores when needed.

5.6 Actions with parameters

A parameter is a value given to an action when the action is called.

Parameters allow one action to work with different values.

The recommended INTENTO Core 0.1 form is:

to action_name with parameter:

    instruction

action_name with value

Example:

to greet with name:

    show "Hello " plus name

greet with "Aurora"

greet with "Alessandro"

Output:

Hello Aurora

Hello Alessandro

The action `greet` has one parameter, `name`. When the program calls `greet with "Aurora"`, the value `"Aurora"` is used inside the action as `name`. When the program calls `greet with "Alessandro"`, the value changes for that call.

The action definition is reusable because it describes the structure of the action, not only one fixed case.

A parameter must be provided when the action expects one.

Invalid:

to greet with name:

    show "Hello " plus name

greet

Expected error:

Semantic error: action `greet` needs 1 parameter.

Correct:

```
to greet with name:
    show "Hello " plus name

greet with "Aurora"
```

Output:

Hello Aurora

Actions may also accept more than one parameter.

```
to show_profile with name and role:
    show name plus " — " plus role

show_profile with "Ada" and "mathematician"
show_profile with "Grace" and "computer scientist"
```

Output:

Ada — mathematician

Grace — computer scientist

The order of parameters matters. The first value goes to the first parameter, the second value goes to the second parameter, and so on.

5.7 Returning values with `return`

Some actions only perform instructions, such as showing text. Other actions should produce a value that the program can use.

The `return` instruction sends a value back from an action.

Example:

```
to make_full_name with first_name and last_name:
```

```
    return first_name plus " " plus last_name


remember full_name as make_full_name with "Ada" and "Lovelace"

show full_name
```

Output:

```
Ada Lovelace
```

The action `make_full_name` receives two values. It joins them with a space and returns the result. The instruction `remember full_name as make_full_name with "Ada" and "Lovelace"` stores that returned value in `full_name`.

A returning action can be used where an expression is expected.

Example:

```
to label with value:
    return "Value: " plus value


show label with 42
```

Output:

```
Value: 42
```

The `return` instruction ends the action and provides its result.

Invalid:

```
return "Hello"
```

Expected error:

```
Syntax error: `return` can only be used inside a custom action.
```

Another invalid case is using an action as a value when it does not return anything.

```
to greet with name:
    show "Hello " plus name


remember message as greet with "Aurora"
```

show message

Expected error:

Type error: action `greet` does not return a value.

The action `greet` shows output, but it does not return a value. If a value is needed, the action must use `return`.

Correct version:

```
to make_greeting with name:
    return "Hello " plus name


remember message as make_greeting with "Aurora"
show message
```

Output:

Hello Aurora

This distinction keeps action behavior clear. Some actions do something. Other actions produce something. They should not be confused.

5.8 Organizing code with actions

Custom actions help avoid deeply nested or repeated code.

Without an action, a program may repeat the same structure many times:

```
show "Member: Ada"
show "Member: Grace"
show "Member: Linus"
```

Output:

Member: Ada

Member: Grace

Member: Linus

This is valid, but it does not scale well.

A better version uses a list, a loop, and a custom action:

```
remember names as ["Ada", "Grace", "Linus"]


to show_member with name:
    show "Member: " plus name


for each name in names:
    show_member with name
```

Output:

```
Member: Ada
Member: Grace
Member: Linus
```

This version is more flexible. If the list changes, the loop still works. If the output format changes, the action can be changed in one place.

Actions should be named after what they do. A name such as `show_member` is clearer than a name such as `do_it`.

Invalid or unclear action names should be rejected by the Runtime if they break naming rules.

Invalid:

```
to show member with name:
    show name
```

Expected error:

```
Syntax error: action names cannot contain spaces.
```

Correct:

```
to show_member with name:
    show name


show_member with "Ada"
```

Output:

Ada

Readable names are not only a style preference. They are part of making INTENTO programs understandable.

5.9 A complete structured example

This example combines lists, `for each`, custom actions, parameters, and `return`.

```
show "Skill report"


remember skills as ["Linux", "Python", "INTENTO"]


to format_skill with skill:
    return "- " plus skill


for each skill in skills:
    remember line as format_skill with skill
    show line


show "Report complete"
```

Output:

```
Skill report
- Linux
- Python
- INTENTO
Report complete
```

The list `skills` contains three text values. The action `format_skill` receives one skill and returns a formatted text line. The loop processes each skill in the list. For every item, the program creates a line and shows it.

The example is small, but it shows an important pattern. Data is stored in a list. Repetition is handled by `for each`. Formatting is isolated inside a custom action. The main program remains readable.

A less organized program could produce the same output by writing every line manually, but it would be harder to change later.

This is the practical reason for repetition and custom actions: they keep the program clear as it grows.

5.10 Common errors

A common error is forgetting the colon after `repeat`.

Invalid:

repeat 3 times

    show "Hello"

Expected error:

Syntax error: expected `:` after `repeat`.

Correct:

repeat 3 times:

    show "Hello"

Output:

Hello

Hello

Hello

Another common error is using `for each` with a value that is not a list.

Invalid:

remember item as "Linux"

for each value in item:

    show value

Expected error:

Type error: `for each` requires a list after `in`.

Correct:

```
remember items as ["Linux"]


for each value in items:
    show value
```

Output:

```
Linux
```

Another common error is calling an unknown action.

Invalid:

```
greet with "Aurora"
```

Expected error:

```
Semantic error: unknown action `greet`.
```

Correct:

```
to greet with name:
    show "Hello " plus name


greet with "Aurora"
```

Output:

```
Hello Aurora
```

Another common error is returning outside an action.

Invalid:

```
return "Done"
```

Expected error:

```
Syntax error: `return` can only be used inside a custom action.
```

These errors should be direct and understandable. A good Runtime should not only reject invalid code; it should tell the programmer what structure is missing or misused.

5.11 Summary

The `repeat` instruction runs a block a fixed number of times. It is useful when the same action must happen more than once.

Lists store multiple values in order. The `for each` instruction processes each item in a list and runs a block for every item.

Custom actions are defined with `to`. They allow the programmer to name a reusable block of behavior. Actions may receive parameters with `with`, and they may produce values with `return`.

Together, these tools make INTENTO programs more structured. Instead of copying instructions, the programmer can express a pattern once and reuse it.

The principle remains the same as in the rest of the language: the code should be readable for a human and precise for the Runtime.

Repetition should not make the program noisy. Lists should make collections clear. Actions should give names to intentions.

Chapter 6 — Files, Safety, and Python-Backed Power

INTENTO is designed to be simple on the surface, but it is not limited to printing messages and storing values.

A real programming language must eventually interact with the world outside the program. It may need to read a file, create a folder, write a note, process data, prepare a project structure, or call a more powerful backend operation.

This is where INTENTO must be especially careful.

A language that can affect files, folders, programs, or external systems must not behave vaguely. It must be readable, but also safe. It must make dangerous actions visible. It must refuse unclear operations. It must ask for confirmation when an operation can change the user’s environment.

This chapter introduces the practical execution layer of INTENTO: file and folder operations, confirmation syntax, safety rules, and Python-backed power.

The central idea is simple: INTENTO should let the user express real intentions, but the Runtime must execute them with clear limits.

6.1 Paths and file values

File and folder operations need paths.

A path tells the Runtime where something is located.

Example:

"notes.txt"

This path refers to a file named `notes.txt`.

Another example:

"Projects/INTENTO"

This path refers to a folder named `INTENTO` inside a folder named `Projects`.

In INTENTO Core 0.1, paths should be written as text values. This means they must be placed between quotation marks when written directly.

Valid:

show "Projects/INTENTO"

Output:

Projects/INTENTO

Invalid:

show Projects/INTENTO

Expected error:

Syntax error: path text must be written between quotation marks.

A path may also come from memory or from an expression that produces text.

Example:

remember folder as "Projects/INTENTO"

show folder

Output:

Projects/INTENTO

This matters because file operations often use dynamic paths. A program may ask for a project name, build a folder path, and then create that folder.

However, paths are sensitive. A Runtime should not allow every possible path by default. In a safe configuration, INTENTO should work inside an allowed workspace, such as the current project folder or a directory explicitly approved by the user.

This protects the system from accidental or unsafe operations.

6.2 Creating folders

A folder can be created with `create folder`.

Basic form:

create folder path

Example:

with confirmation:

```
create folder "Projects/INTENTO"


show "Folder ready"
```

Expected confirmation prompt:

Confirm operation: create folder Projects/INTENTO?

If the user confirms, output:

Folder ready

Expected filesystem result:

Projects/INTENTO/

The `create folder` instruction changes the filesystem, so it should normally run inside a `with confirmation` block unless the Runtime configuration explicitly marks the target as safe.

The instruction should be predictable. If the folder does not exist, the Runtime creates it. If the folder already exists, the Runtime may treat the operation as already satisfied instead of failing.

Example:

with confirmation:

```
create folder "Projects/INTENTO"


show "Folder checked"
```

Possible output when the folder already exists:

Folder checked

Possible execution note:

Folder already exists: Projects/INTENTO

The exact execution note may depend on the Runtime, but the program should not behave destructively. Creating a folder should not erase an existing folder.

Invalid path:

with confirmation:

```
    create folder "/system/private"
```

Expected error:

Safety error: path is outside the allowed workspace.

The purpose of this error is not to limit the language unnecessarily. It is to make sure that INTENTO programs cannot accidentally modify sensitive parts of the system.

6.3 Creating files

A file can be created with `create file`.

Basic form:

create file path with content

The path must produce text. The content must also produce a value that can be written as text.

Example:

with confirmation:

```
    create file "notes.txt" with "Hello from INTENTO"


show "File created"
```

Expected confirmation prompt:

Confirm operation: create file notes.txt?

If the user confirms, output:

File created

Expected filesystem result:

notes.txt

Expected file content:

Hello from INTENTO

In INTENTO Core 0.1, `create file` should be non-destructive by default. If the file already exists, the Runtime should not overwrite it silently.

Example:

with confirmation:

```
create file "notes.txt" with "New content"
```

Expected error if `notes.txt` already exists:

Safety error: file already exists. Use an explicit overwrite operation if replacement is intended.

This rule is important. A human-readable language should not hide destructive behavior behind friendly syntax. Creating a file and overwriting a file are different intentions, so they should use different rules.

A program may build the file content using expressions:

remember title as "INTENTO"

remember line as "Project: " plus title

with confirmation:

```
create file "project.txt" with line
```

show "Project file created"

Expected confirmation prompt:

Confirm operation: create file project.txt?

If the user confirms, output:

Project file created

Expected file content:

Project: INTENTO

The file content is calculated before the file is created. The Runtime stores the result of `"Project: " plus title` in `line`, then writes that value into the file.

6.4 Reading files

Reading a file does not modify the filesystem. For this reason, reading is usually safer than writing.

A file can be read with `read file`.

Basic form:

read file path and call it name

Example:

read file "notes.txt" and call it notes

show notes

Expected output if `notes.txt` contains `Hello from INTENTO`:

Hello from INTENTO

The instruction reads the file content and stores it under the name `notes`. After that, the program can use `notes` like any other remembered value.

If the file does not exist, the Runtime should report a clear error.

read file "missing.txt" and call it content

show content

Expected error:

File error: file not found: missing.txt.

Reading should still obey the Runtime safety rules. A program should not be able to read arbitrary sensitive files unless the user or configuration has explicitly allowed that location.

Invalid:

read file "/system/private.txt" and call it secret

show secret

Expected error:

Safety error: path is outside the allowed workspace.

This protects the user’s data. A language that reads files must be clear not only about what it can do, but also about what it refuses to do.

6.5 Appending to files

Sometimes a program should add new content to the end of an existing file.

This can be done with `append to file`.

Basic form:

append to file path with content

Example:

with confirmation:

```
append to file "log.txt" with "Program started"


show "Log updated"
```

Expected confirmation prompt:

Confirm operation: append to file log.txt?

If the user confirms, output:

Log updated

Expected effect:

The line or text "Program started" is added to the end of log.txt.

Appending modifies a file, so it should require confirmation unless the Runtime configuration explicitly allows the target file.

The Runtime should also be clear about line behavior. In INTENTO Core 0.1, a simple rule is recommended: `append to file` adds the given content followed by a newline.

Example:

with confirmation:

```
append to file "log.txt" with "First entry"

append to file "log.txt" with "Second entry"
```

show "Log complete"

Expected confirmation prompt:

Confirm operation: append to file log.txt?

Confirm operation: append to file log.txt?

If the user confirms both operations, output:

Log complete

Expected file content:

First entry

Second entry

If the file does not exist, the Runtime should not silently decide what to do unless the behavior is defined. In Core 0.1, the safer rule is to report an error.

with confirmation:

    append to file "missing-log.txt" with "Entry"

Expected error:

File error: cannot append because file does not exist: missing-log.txt.

A future Runtime may offer an explicit instruction such as `create file if missing`, but Core syntax should remain clear and conservative.

6.6 Confirmation with `with confirmation`

The `with confirmation` block marks operations that need user approval before execution.

Basic form:

with confirmation:

    instruction

Example:

with confirmation:

    create folder "Projects/Test"

show "Done"

Expected confirmation prompt:

Confirm operation: create folder Projects/Test?

If the user confirms, output:

Done

If the user refuses, possible output:

Operation cancelled by user.

Expected program behavior:

The folder is not created.

A confirmation block can contain more than one operation.

with confirmation:

    create folder "Projects/Test"

    create file "Projects/Test/README.txt" with "Test project"

show "Project prepared"

Expected confirmation prompts:

Confirm operation: create folder Projects/Test?

Confirm operation: create file Projects/Test/README.txt?

If the user confirms both operations, output:

Project prepared

Expected filesystem result:

Projects/Test/

Projects/Test/README.txt

Expected file content:

Test project

A Runtime may also choose to summarize the whole block and ask for one confirmation. That is an implementation choice. The essential rule is that the user must clearly know what operation is about to affect the system.

The `with confirmation` block should not be treated as decoration. It is part of the safety model of INTENTO.

6.7 Safe, confirmed, and blocked operations

INTENTO should classify operations before executing them.

Some operations are safe because they only affect program memory or output. Examples include `show`, `remember`, simple expressions, and most comparisons.

Some operations require confirmation because they modify the user's environment. Creating folders, creating files, appending to files, overwriting files, moving files, or calling external services should normally be confirmed.

Some operations should be blocked completely in Core 0.1 because they are too dangerous, too vague, or too dependent on the operating system.

For example, raw shell execution should not be allowed by default.

Invalid:

run shell "some system command"

Expected error:

Safety error: raw shell execution is not allowed in INTENTO Core.

The exact text inside the shell command is not important. The problem is the instruction itself. Raw shell execution gives too much power without enough structure.

Another blocked case is a path outside the allowed workspace:

with confirmation:

    create file "/system/config.txt" with "change"

Expected error:

Safety error: path is outside the allowed workspace.

Confirmation does not automatically make every action acceptable. Some actions should remain blocked even if the user wrote them inside `with confirmation`.

This is an important principle:

confirmation is not a replacement for safety rules

The Runtime must still refuse operations that violate its safety policy.

6.8 Overwriting and deletion

Overwriting and deletion are more dangerous than creation or reading because they can destroy information.

For this reason, INTENTO Core 0.1 should be conservative.

A basic Runtime may choose not to support deletion at all in Core 0.1. If deletion is introduced later, it should require explicit syntax, clear confirmation, and strong path restrictions.

Invalid in Core 0.1:

delete file "notes.txt"

Expected error:

Safety error: deletion is not available in INTENTO Core 0.1.

Overwriting should also be explicit. The instruction `create file` should not overwrite an existing file. If a future Runtime supports overwriting, it should use a clear instruction such as:

overwrite file path with content

That operation should always require confirmation.

Possible future syntax:

with confirmation:

    overwrite file "notes.txt" with "Replacement text"

Possible confirmation prompt:

Confirm operation: overwrite file notes.txt?

Possible output if confirmed:

(no visible output)

Because overwriting can destroy previous content, a Runtime may also create backups automatically or require an additional safety setting. The exact policy belongs to the Runtime, but the language principle is clear: destructive actions must never be hidden behind harmless-looking words.

6.9 Python-backed actions

INTENTO is designed to remain human-readable, but it can rely on Python as a power layer underneath.

This does not mean that INTENTO programs should contain raw Python code. Instead, the Runtime may expose registered Python-backed actions with simple INTENTO syntax.

A Python-backed action is a safe, named capability implemented in Python and made available to INTENTO.

Example:

```
remember title as "My First Project"


use python action "slugify" with title and call the result slug


show slug
```

Output:

```
my-first-project
```

The INTENTO program does not define how slugification works internally. The Python backend does that. INTENTO only expresses the intention: use the registered Python action called `"slugify"`, give it the value `title`, and store the result as `slug`.

Another example:

```
remember text as "INTENTO is simple and powerful"


use python action "count_words" with text and call the result count


show "Words: " plus count
```

Output:

```
Words: 5
```

The Python action `count_words` receives the text and returns a number. INTENTO stores that number as `count`.

The important rule is that Python-backed actions must be registered and controlled. The Runtime should know which Python actions exist, what types of values they accept, what they return, and whether they require confirmation.

Invalid:

use python code "print('hello')"

Expected error:

Safety error: raw Python code is not allowed in INTENTO programs.

This keeps the separation clean. Python provides power. INTENTO provides readable intention. The user writes controlled INTENTO code, not arbitrary backend code.

6.10 Python power without losing INTENTO identity

Python-backed actions can make INTENTO much more capable.

They may support operations such as text processing, data conversion, CSV parsing, JSON handling, image processing, speech synthesis, network requests, database access, or integration with external tools. But each capability should be exposed through a clear INTENTO instruction or a registered action.

For example, a future Runtime may provide this:

read file "notes.txt" and call it notes

use python action "count_words" with notes and call the result count

show "Word count: " plus count

Expected output if `notes.txt` contains `one two three`:

Word count: 3

The program remains readable. The Python backend does the technical work, but the INTENTO source still describes the intention clearly.

A bad design would expose Python as a hidden escape hatch where anything can happen. That would make INTENTO powerful, but unsafe and less understandable.

A better design is to expose Python through named, typed, documented actions.

This means every Python-backed action should answer these questions:

What is the action called?

What values does it accept?

What value does it return?

Does it read files?

Does it write files?

Does it require confirmation?

Can it access the network?

The user should not have to inspect Python code to understand what an INTENTO program is trying to do.

6.11 A complete file and Python-backed example

This example creates a small project folder, creates a README file, and uses a Python-backed action to create a safe folder name.

```
show "Project setup"


ask "Project title?" and call the answer title


if title is empty:
    show "Project title is required"
    stop


use python action "slugify" with title and call the result folder_name


remember folder_path as "Projects/" plus folder_name
remember readme_path as folder_path plus "/README.txt"
remember readme_text as "# " plus title


with confirmation:
    create folder folder_path
    create file readme_path with readme_text
```

show "Project created: " plus folder_path

Example interaction:

Project setup

Project title?

My First Project

Confirm operation: create folder Projects/my-first-project?

Confirm operation: create file Projects/my-first-project/README.txt?

Project created: Projects/my-first-project

Expected filesystem result:

Projects/my-first-project/

Projects/my-first-project/README.txt

Expected README content:

My First Project

This example shows the full INTENTO idea in a compact form.

The program asks for information. It validates the answer. It uses a Python-backed action to transform the title into a safe folder name. It builds paths using readable expressions. It places file operations inside `with confirmation`. Finally, it shows the result.

The Runtime does the technical work, but the source code remains understandable.

6.12 Common errors

A common error is writing a file path without quotation marks.

Invalid:

read file notes.txt and call it notes

Expected error:

Syntax error: file path must be text.

Correct:

read file "notes.txt" and call it notes

show notes

Expected output if `notes.txt` contains `Hello`:

Hello

Another common error is trying to write a file without confirmation.

Invalid in a strict Runtime:

create file "notes.txt" with "Hello"

Expected error:

Safety error: file creation requires confirmation.

Correct:

with confirmation:

    create file "notes.txt" with "Hello"

show "File created"

Expected confirmation prompt:

Confirm operation: create file notes.txt?

If the user confirms, output:

File created

Another common error is using an unknown Python-backed action.

Invalid:

remember text as "hello"

use python action "unknown_action" with text and call the result value

show value

Expected error:

Semantic error: unknown Python action `unknown_action`.

Correct, if the Runtime provides `count_words`:

remember text as "hello world"

use python action "count_words" with text and call the result count

show count

Output:

2

Another common error is assuming that confirmation allows blocked behavior.

Invalid:

with confirmation:

    run shell "some system command"

Expected error:

Safety error: raw shell execution is not allowed in INTENTO Core.

The Runtime should reject this even inside a confirmation block. Confirmation is necessary for many sensitive actions, but it does not override the safety model.

6.13 Summary

INTENTO can interact with files, folders, and backend capabilities, but it must do so safely.

Paths are text values or expressions that produce text. File operations must be clear about what they do. Reading is usually safe, while creating, appending, overwriting, or deleting require stronger protection.

The `with confirmation` block makes sensitive operations visible to the user before they happen. It is an essential part of INTENTO's safety syntax.

The Runtime should classify operations as safe, confirmation-required, or blocked. Some actions should remain blocked in Core 0.1 even if the user asks for confirmation.

Python-backed actions allow INTENTO to become powerful without losing its identity. Python can perform complex work underneath, but INTENTO remains the readable layer where the user expresses intention.

The final rule of the manual is the same as the first rule of the language:

human-readable does not mean uncontrolled

INTENTO should be simple to read, precise to parse, safe to execute, and powerful enough to grow.

INTENTO — Full Language Extension

Additional Chapters

Chapter 7 — Advanced Expressions, Operators, and Conversion

This chapter completes the expression system of INTENTO. It introduces operators beyond `plus`, such as subtraction, multiplication, division, grouping, and precedence. It also defines explicit type conversion between text, numbers, booleans, and displayable values. The goal is to make INTENTO more powerful without allowing the Runtime to guess silently.

Chapter 8 — Errors, Scope, and Runtime Rules

This chapter defines how INTENTO behaves when something goes wrong. It explains syntax errors, semantic errors, type errors, file errors, safety errors, runtime errors, and action errors. It also defines scope: where remembered values exist, how parameters behave inside custom actions, what happens after `return`, and when execution should stop.

Chapter 9 — Modules, Libraries, and Registered Actions

This chapter explains how INTENTO programs can grow beyond a single file. It introduces modules, reusable libraries, imports, namespaces, registered actions, and safe extension points. It also clarifies how Python-backed actions are registered, typed, documented, and exposed to INTENTO without allowing uncontrolled backend execution.

Chapter 10 — Standard Library and Real Programs

This chapter presents the practical standard library of INTENTO and shows complete real-world programs. It covers common tools for text, numbers, lists, files, dates, JSON, CSV, logs, and controlled integrations. It closes the manual by showing INTENTO as a complete language: readable, safe, extensible, and executable.

Chapter 7 — Advanced Expressions, Operators, and Conversion

INTENTO Core begins with one basic expression operator: `plus`.

That is enough to introduce calculation and text joining, but a complete language needs more. Programs must subtract values, multiply, divide, group expressions, control evaluation order, and convert values from one type to another when needed.

This chapter extends the expression system while keeping the main principle of INTENTO intact: expressions should be readable for humans and precise for the Runtime.

The goal is not to imitate symbolic programming languages immediately. INTENTO can support symbols later, but its official readable form should remain clear:

plus

minus

times

divided by

These words are longer than symbols, but they make the intention visible.

7.1 Expressions produce values

An expression is any piece of code that produces a value.

A simple value is an expression:

show 10

Output:

10

A remembered name is also an expression:

remember price as 30

show price

Output:

30

A combined expression uses operators:

remember price as 30

remember tax as 5

show price plus tax

Output:

35

The Runtime evaluates the expression first. Then the instruction uses the result. In this example, `price plus tax` becomes `35`, and `show` displays `35`.

The same rule applies when an expression is stored:

remember price as 30

remember tax as 5

remember total as price plus tax

show total

Output:

35

The value stored in `total` is the result of the expression, not the expression text itself.

7.2 Numeric operators

INTENTO extended syntax defines four basic numeric operators:

plus

minus

times

divided by

When used with numbers, they perform arithmetic.

Example:

show 10 plus 5

show 10 minus 5

show 10 times 5

show 10 divided by 5

Output:

15

5

50

2

Each expression produces a number.

These operators can be used with remembered values:

remember price as 30

remember discount as 5

remember quantity as 2

remember reduced_price as price minus discount

remember total as reduced_price times quantity

show "Reduced price: " plus reduced_price

show "Total: " plus total

Output:

Reduced price: 25

Total: 50

The program first calculates the reduced price, then multiplies it by the quantity. Naming intermediate results makes the program easier to read.

Numeric operators should require numeric values. This is invalid:

show "10" minus 5

Expected error:

Type error: `minus` requires numbers.

The value `"10"` is text, not a number. If the programmer wants to calculate with it, it must be converted explicitly.

7.3 Division

The operator `divided by` divides one number by another.

Example:

show 10 divided by 2

Output:

5

If the result is not a whole number, the Runtime may return a decimal value.

show 5 divided by 2

Output:

2.5

Division by zero must be rejected.

Invalid:

show 10 divided by 0

Expected error:

Runtime error: division by zero.

This is a runtime error because the expression is syntactically valid, but it cannot be executed safely.

A program can avoid this error by checking the divisor before dividing:

```
remember total as 10
remember divisor as 0


if divisor is 0:
    show "Cannot divide by zero"
else:
    show total divided by divisor
```

Output:

Cannot divide by zero

This is good INTENTO style: do not let an invalid operation happen when the program can check the value first.

7.4 Expression order and precedence

When an expression contains more than one operator, the Runtime must know which operation to evaluate first.

INTENTO should use a simple and familiar precedence rule:

parentheses first

times and divided by before plus and minus

operators with the same priority from left to right

Example:

show 2 plus 3 times 4

Output:

14

The multiplication happens first. The expression is interpreted as:

2 plus (3 times 4)

So the result is:

2 plus 12 = 14

If the programmer wants the addition to happen first, parentheses should be used:

show (2 plus 3) times 4

Output:

20

Parentheses make the intention explicit. They are also useful for readability.

Another example:

remember price as 30

remember tax as 5

remember quantity as 2

show (price plus tax) times quantity

Output:

70

The expression first calculates `price plus tax`, producing `35`. Then it multiplies the result by `quantity`, producing `70`.

Without parentheses, the result would be different:

remember price as 30

remember tax as 5

remember quantity as 2

show price plus tax times quantity

Output:

40

Here, `tax times quantity` is evaluated first. The result is `10`, then `price plus 10` becomes `40`.

Both programs are valid. They express different meanings.

7.5 Parentheses

Parentheses group expressions.

They do not produce output by themselves. They only tell the Runtime how to evaluate the expression.

Example:

show (10 plus 5) times 2

Output:

30

The expression inside parentheses is evaluated first. Then the result is multiplied by `2`.

Parentheses can also make expressions easier to read even when they are not strictly required.

Example:

remember first as 10

remember second as 5

remember third as 2

show first plus (second times third)

Output:

20

The parentheses make the intended grouping visible. A reader does not need to remember precedence rules to understand the expression.

Invalid parentheses should produce clear syntax errors.

Invalid:

show (10 plus 5

Expected error:

Syntax error: missing closing parenthesis.

Invalid:

show 10 plus 5)

Expected error:

Syntax error: unexpected closing parenthesis.

Good error messages are especially important in expressions because a small missing symbol can change or break the whole line.

7.6 Text joining with `plus`

The operator `plus` has two defined roles.

When both values are numbers, it performs numeric addition.

show 10 plus 5

Output:

15

When text is involved, it joins values into text.

remember name as "Aurora"

show "Hello " plus name

Output:

Hello Aurora

This rule makes output messages simple to write.

Example:

remember product as "Notebook"

remember price as 30

show "Product: " plus product

show "Price: " plus price

Output:

Product: Notebook

Price: 30

The number `price` is converted to display text only while building the output message. The stored value remains a number.

This can be verified:

remember price as 30

show "Price: " plus price

show price plus 5

Output:

Price: 30

35

The first `show` joins the number into a text message. The second `show` performs numeric addition.

This distinction keeps INTENTO readable without making types disappear.

7.7 Text that looks numeric

Text that looks like a number is still text.

Example:

show "10" plus "5"

Output:

105

The result is text joining, not numeric addition.

The numeric version is:

show 10 plus 5

Output:

15

This rule is essential because user input often arrives as text. If a user types `10`, the visible characters look numeric, but the program should not silently guess the type.

Invalid:

remember first as "10"

remember second as 5

show first minus second

Expected error:

Type error: `minus` requires numbers.

The fix is explicit conversion.

7.8 Explicit conversion

Conversion means turning a value from one type into another.

INTENTO should make conversion visible. The Runtime should not silently change types in hidden ways.

The recommended syntax is:

convert value to type and call it name

Example:

remember text_age as "42"

```
convert text_age to number and call it age


show age plus 1
```

Output:

```
43
```

The first value, `text_age`, is text. The conversion instruction creates a new value, `age`, as a number. After that, numeric operations are valid.

If conversion fails, the Runtime should report a clear error.

Invalid:

```
remember text_age as "forty-two"


convert text_age to number and call it age


show age plus 1
```

Expected error:

```
Conversion error: cannot convert `forty-two` to number.
```

Conversion can also turn numbers into text.

```
remember age as 42


convert age to text and call it age_text


show "Age as text: " plus age_text
```

Output:

```
Age as text: 42
```

In this example, conversion is not strictly necessary for display joining, because `show "Age: " plus age` already works. But explicit conversion is useful when the program wants to store the text version for later use.

7.9 Converting user input

User input should usually be treated as text until converted.

Example:

```
ask "Enter your age:" and call the answer age_text


if age_text is empty:
    show "Age is required"
    stop


convert age_text to number and call it age


show "Next year you will be " plus (age plus 1)
```

Example interaction:

```
Enter your age:
42
Next year you will be 43
```

This program is clear because it separates the steps.

First, it asks for input. Then it checks whether the input is empty. Then it converts the text to a number. Only after conversion does it perform numeric addition.

If the user enters invalid text, conversion fails:

```
Enter your age:
abc
Conversion error: cannot convert `abc` to number.
```

A more advanced Runtime may allow the program to catch and handle conversion errors. That belongs to the error-handling rules defined in the next chapter. For now, the important point is that conversion is explicit and visible.

7.10 Boolean conversion

Boolean conversion should be strict.

The text values `"true"` and `"false"` may be converted to boolean values.

Example:

```
remember answer as "true"

convert answer to boolean and call it active

if active is true:
    show "Active"
else:
    show "Not active"
```

Output:

```
Active
```

The text `"true"` becomes the boolean `true`.

The same works with `"false"`:

```
remember answer as "false"

convert answer to boolean and call it active

if active is true:
    show "Active"
else:
    show "Not active"
```

Output:

```
Not active
```

Other text should not be guessed.

Invalid:

remember answer as "yes"

convert answer to boolean and call it active

Expected error:

Conversion error: cannot convert `yes` to boolean.

A future standard library may provide higher-level helpers that interpret values such as `"yes"`, `"no"`, `"y"`, or `"n"`, but the core conversion should remain strict.

Strict conversion prevents surprising behavior.

7.11 Symbolic aliases

INTENTO’s official readable operators are word-based:

plus

minus

times

divided by

A future Runtime may allow symbolic aliases:

+

-

*

/

For example, this may be accepted by some Runtime versions:

show 10 + 5

Possible output:

15

However, the official manual form remains:

show 10 plus 5

Output:

15

This is not only an aesthetic choice. The readable form protects INTENTO’s identity. Symbols may be useful for experienced users, but the language should remain understandable for beginners and non-programmers.

If symbolic aliases are supported, they must follow the same rules as the word operators. They should not introduce different behavior.

For example:

show (2 + 3) * 4

Possible output:

20

Equivalent official form:

show (2 plus 3) times 4

Output:

20

The symbols are shorter. The word form is clearer. The Runtime may support both, but this manual continues to use the readable form.

7.12 Common expression errors

One common error is mixing text and numeric operators.

Invalid:

remember value as "10"

show value times 2

Expected error:

Type error: `times` requires numbers.

Correct:

remember value as "10"

convert value to number and call it number_value

show number_value times 2

Output:

20

Another common error is expecting quoted expressions to be calculated.

show "5 plus 7"

Output:

5 plus 7

This is not an error. It is literal text. The Runtime does not calculate inside quotation marks.

Correct calculation:

show 5 plus 7

Output:

12

Another error is relying on the wrong evaluation order.

show 2 plus 3 times 4

Output:

14

If the programmer wanted `2 plus 3` first, parentheses are required:

show (2 plus 3) times 4

Output:

20

Another common error is dividing by zero.

Invalid:

show 5 divided by 0

Expected error:

Runtime error: division by zero.

Expressions are powerful, but they must remain readable. When in doubt, use remembered names and parentheses.

7.13 A complete calculation example

This example uses arithmetic, grouping, conversion, and readable output.

```
show "Invoice calculator"

ask "Base price?" and call the answer base_text

if base_text is empty:
    show "Base price is required"
    stop

convert base_text to number and call it base_price

remember tax as 5
remember shipping as 10
remember subtotal as base_price plus tax
remember total as subtotal plus shipping

show "Base price: " plus base_price
show "Tax: " plus tax
show "Shipping: " plus shipping
show "Total: " plus total
```

Example interaction:

```
Invoice calculator
Base price?
```

30

Base price: 30

Tax: 5

Shipping: 10

Total: 45

The program asks for a price, validates that the answer is not empty, converts the answer to a number, calculates the subtotal and total, then displays the result.

If the user enters invalid text, the program fails at conversion:

Invoice calculator

Base price?

abc

Conversion error: cannot convert `abc` to number.

This is correct. The Runtime refuses to calculate with invalid numeric data.

The program remains readable because each important result has a name. Instead of hiding everything inside one long expression, it uses memory to make the intention visible.

7.14 Summary

Advanced expressions make INTENTO more capable.

The basic numeric operators are:

plus

minus

times

divided by

The operator `plus` can add numbers or join text. The operators `minus`, `times`, and `divided by` require numbers.

Parentheses control evaluation order. Without parentheses, multiplication and division happen before addition and subtraction.

Conversion must be explicit. The recommended syntax is:

convert value to type and call it name

This allows programs to convert user input or stored values safely.

Symbols such as `+`, `-`, `*`, and `/` may become optional aliases, but the official INTENTO form remains word-based because it is more readable.

The main rule of this chapter is simple:

expressions should be powerful, but never hidden or ambiguous

A good INTENTO expression says what it means, and the Runtime evaluates it without guessing.

Chapter 8 — Errors, Scope, and Runtime Rules

A programming language is not complete only when it can run correct programs. It also needs to behave clearly when something goes wrong.

Errors are not an accessory part of the language. They are part of the language design. A good Runtime should not fail mysteriously, continue in an unsafe state, or guess what the programmer meant. It should explain the problem, identify the kind of error, and stop execution when continuing would be unclear or unsafe.

This chapter defines how INTENTO handles errors, how remembered values exist in memory, how custom actions create their own scope, and what rules the Runtime should follow during execution.

The central principle is simple: INTENTO should be friendly to read, but strict when meaning is unclear.

8.1 Why errors matter

An INTENTO program is written for both humans and the Runtime.

The human reader needs clarity. The Runtime needs precision. When the source code breaks a rule, the Runtime should not pretend that everything is fine.

Example:

```
show "Before"

show unknown_value

show "After"
```

Expected error:

```
Semantic error: unknown name `unknown_value`.
```

The name `unknown_value` was never remembered. The Runtime should not invent a value, display an empty line, or treat the name as literal text. The program is invalid because the meaning is not defined.

A good error protects the programmer. It says, in effect: “I cannot safely execute this because a required meaning is missing.”

Errors are especially important in INTENTO because the language is designed to look natural. Natural-looking code must still obey exact rules. Without strong errors, human-readable syntax becomes vague.

8.2 Main error categories

INTENTO should use clear error categories.

A syntax error means the source code does not follow the grammar of the language.

A semantic error means the structure is valid, but the meaning is not valid. For example, using a name that does not exist.

A type error means a value is used in a way that does not fit its type.

A conversion error means a value cannot be converted to the requested type.

A file error means a file or folder operation cannot be completed.

A safety error means the operation violates the Runtime safety policy.

A runtime error means the program reaches a valid instruction that cannot be completed during execution, such as division by zero.

An action error means a custom action is called incorrectly, returns incorrectly, or is used in the wrong context.

The exact wording of messages may vary between Runtime implementations, but the category and meaning should remain clear.

Example:

show 10 divided by 0

Expected error:

Runtime error: division by zero.

This message is clear. The syntax is valid. The types are valid. The problem appears during execution because division by zero is impossible.

8.3 Syntax errors

A syntax error happens when the Runtime cannot parse the program according to INTENTO grammar.

Example:

```
if name is empty
   show "Name is required"
```

Expected error:

Syntax error: expected `:` after condition.

The problem is structural. The `if` line must end with `:` because it starts a block.

Correct version:

```
remember name as empty

if name is empty:
   show "Name is required"
```

Output:

```
Name is required
```

Another syntax error is missing indentation after a block starter.

Invalid:

```
remember name as empty

if name is empty:
show "Name is required"
```

Expected error:

Syntax error: expected an indented block after `if`.

Correct version:

```
remember name as empty
```

if name is empty:

    show "Name is required"

Output:

Name is required

Syntax errors should usually stop the program before execution begins. If the source file cannot be parsed, the Runtime should not run part of it and ignore the rest. The program structure must be valid before the program can safely execute.

8.4 Semantic errors

A semantic error happens when the code has a valid structure, but the meaning is invalid.

The most common semantic error is using a name that does not exist.

Invalid:

show project

Expected error:

Semantic error: unknown name `project`.

The instruction has a valid shape: `show` followed by a value or name. The problem is that `project` was never remembered.

Correct version:

remember project as "INTENTO"

show project

Output:

INTENTO

Another semantic error is calling an action that has not been defined.

Invalid:

greet with "Aurora"

Expected error:

Semantic error: unknown action `greet`.

Correct version:

to greet with name:

    show "Hello " plus name

greet with "Aurora"

Output:

Hello Aurora

Semantic errors should be detected as early as possible. If the Runtime can know before execution that a name or action is missing, it should report the error before running the program. This prevents partial execution of a program that is already known to be invalid.

8.5 Type errors

A type error happens when a value is used with an operation that does not fit its type.

Example:

show true plus false

Expected error:

Type error: cannot use `plus` with boolean and boolean.

The values `true` and `false` are booleans. They are not numbers to add, and they are not text values to join. The Runtime should not guess a meaning.

Another example:

remember value as "10"

show value times 2

Expected error:

Type error: `times` requires numbers.

The value `"10"` is text, not a number. The correct program must convert it first.

remember value as "10"

convert value to number and call it number_value

show number_value times 2

Output:

20

Type errors keep programs predictable. They prevent the Runtime from silently changing the meaning of values.

8.6 Conversion errors

A conversion error happens when the program asks the Runtime to convert a value, but the value cannot be converted safely.

Example:

```
remember age_text as "42"

convert age_text to number and call it age

show age plus 1
```

Output:

43

This works because the text `"42"` can be converted to the number `42`.

Now compare this:

```
remember age_text as "forty-two"

convert age_text to number and call it age

show age plus 1
```

Expected error:

Conversion error: cannot convert `forty-two` to number.

The Runtime should not guess that `"forty-two"` means `42`. The conversion rule must be precise.

Boolean conversion should also be strict.

remember answer as "true"

convert answer to boolean and call it active

if active is true:

    show "Active"

else:

    show "Not active"

Output:

Active

Invalid:

remember answer as "yes"

convert answer to boolean and call it active

Expected error:

Conversion error: cannot convert `yes` to boolean.

A future standard library may provide helper actions for flexible human input, such as accepting `"yes"` and `"no"`. Core conversion should remain strict because strict rules are easier to trust.

8.7 Runtime errors

A runtime error happens while the program is executing.

Some problems cannot always be fully known before execution. For example, division by zero may depend on a value entered by the user or calculated during the program.

Example:

show "Before division"

remember total as 10

```
remember divisor as 0


show total divided by divisor


show "After division"
```

Expected output:

```
Before division
```

Runtime error: division by zero.

The final line does not run. Once an unhandled runtime error occurs, execution stops.

The program can avoid the error by checking the value first:

```
remember total as 10

remember divisor as 0


if divisor is 0:
    show "Cannot divide by zero"
else:
    show total divided by divisor
```

Output:

```
Cannot divide by zero
```

This is better style. The program knows the operation may be invalid, so it checks before performing it.

The Runtime should not continue after an unhandled runtime error. Continuing could produce misleading results or unsafe behavior.

8.8 File and safety errors

File operations can fail for practical reasons. A file may not exist, a path may be outside the allowed workspace, or an operation may require confirmation.

Example:

read file "missing.txt" and call it content

show content

Expected error:

File error: file not found: missing.txt.

The syntax is valid, but the file cannot be read because it does not exist.

A safety error is different. It means the requested operation violates the Runtime safety policy.

Invalid:

with confirmation:

    create file "/system/config.txt" with "change"

Expected error:

Safety error: path is outside the allowed workspace.

Even though the operation is inside `with confirmation`, the Runtime must still reject it. Confirmation does not override safety rules.

Another invalid example:

with confirmation:

    run shell "some system command"

Expected error:

Safety error: raw shell execution is not allowed in INTENTO Core.

Safety errors should stop execution. A program that attempts a blocked operation should not continue as if nothing happened, because the rest of the program may depend on that operation having succeeded.

8.9 Action errors

Custom actions must be called with the correct number of parameters.

Invalid:

to greet with name:

    show "Hello " plus name

greet

Expected error:

Action error: action `greet` needs 1 parameter.

Correct:

to greet with name:

    show "Hello " plus name

greet with "Aurora"

Output:

Hello Aurora

Another action error happens when an action is used as a value but does not return anything.

Invalid:

to greet with name:

    show "Hello " plus name

remember message as greet with "Aurora"

show message

Expected error:

Action error: action `greet` does not return a value.

Correct:

to make_greeting with name:

    return "Hello " plus name

remember message as make_greeting with "Aurora"

show message

Output:

Hello Aurora

The difference is important. An action that uses `show` produces output. An action that uses `return` produces a value. These are not the same thing.

The `return` instruction can only be used inside a custom action.

Invalid:

return "Done"

Expected error:

Syntax error: `return` can only be used inside a custom action.

This prevents unclear program flow.

8.10 Scope

Scope defines where a remembered value exists and where it can be used.

At the top level of a program, remembered values belong to the main program memory.

Example:

remember name as "Aurora"

show name

Output:

Aurora

The value `name` exists after it is remembered and can be used by later top-level instructions.

Blocks such as `if`, `else`, `repeat`, and `with confirmation` use the current scope. If a remembered value is created inside a top-level block and that block runs, the value becomes available afterward.

Example:

remember name as empty

if name is empty:

    remember message as "Name is missing"

show message

Output:

Name is missing

The block runs, so `message` is remembered.

If the block does not run, the value is not created.

```
remember name as "Aurora"


if name is empty:
    remember message as "Name is missing"


show message
```

Expected error:

Semantic error: unknown name `message`.

A safer style is to create a default value before the condition:

```
remember name as "Aurora"

remember message as "Name is present"


if name is empty:
    remember message as "Name is missing"


show message
```

Output:

Name is present

This makes the program easier to reason about because `message` always exists before it is shown.

8.11 Action scope

Custom actions create their own local scope.

Parameters exist inside the action. Values remembered inside the action also exist inside that action. When the action finishes, its local memory disappears.

Example:

```
to make_greeting with name:
    remember message as "Hello " plus name
    return message

remember result as make_greeting with "Aurora"
show result
```

Output:

```
Hello Aurora
```

Inside the action, `name` is a parameter and `message` is a local remembered value. The action returns `message`, so the caller receives the final text as `result`.

The local value `message` should not be available outside the action.

Invalid:

```
to make_greeting with name:
    remember message as "Hello " plus name
    return message

remember result as make_greeting with "Aurora"

show message
```

Expected error:

```
Semantic error: unknown name `message`.
```

This rule prevents accidental dependency on internal action details. The caller should use the returned value, not the action’s private memory.

Actions should receive the values they need through parameters. They should not depend on hidden access to outer memory.

Invalid:

```
remember prefix as "Hello "



to greet with name:

    show prefix plus name



greet with "Aurora"
```

Expected error:

```
Semantic error: unknown name `prefix` in action `greet`.
```

Correct:

```
to greet with prefix and name:

    show prefix plus name



greet with "Hello " and "Aurora"
```

Output:

```
Hello Aurora
```

This makes dependencies visible. Anyone reading the action call can see which values are being passed into the action.

8.12 `return` and action flow

The `return` instruction ends the current action and sends a value back to the caller.

Example:

```
to check_name with name:

    if name is empty:

        return "Name is missing"
```

```
    return "Name is present"
```

```
show check_name with empty

show check_name with "Aurora"
```

Output:

```
Name is missing

Name is present
```

When `name` is empty, the first `return` runs and the action ends immediately. The second `return` is not reached.

This means that instructions after a successful `return` in the same path do not run.

Example:

```
to test:

    return "Finished"

    show "This will not run"


show test
```

Output:

```
Finished
```

The line after `return` is unreachable in normal execution. A Runtime may warn about unreachable code, but it does not need to treat it as a fatal error.

A returning action should return a value on every path when it is used as an expression.

Invalid:

```
to maybe_name with value:

    if value is empty:

        return "Missing"
```

remember result as maybe_name with "Aurora"

show result

Expected error:

Action error: action `maybe_name` may finish without returning a value.

Correct:

```
to maybe_name with value:
    if value is empty:
        return "Missing"
    else:
        return value


remember result as maybe_name with "Aurora"
show result
```

Output:

Aurora

This rule makes returning actions reliable. If the caller expects a value, the action must clearly provide one.

8.13 Handling errors with `try` and `on error`

By default, an unhandled error stops the program. This is safe, but some programs need to recover from predictable failures, especially conversion failures or missing optional files.

INTENTO can use a controlled error-handling structure:

```
try:
    instruction
on error:
    instruction
```

The `try` block contains instructions that may fail. The `on error` block runs only if an error happens inside the `try` block.

Example:

```
remember age_text as "abc"


try:
    convert age_text to number and call it age
    show age plus 1
on error:
    show "Age must be a number"
```

Output:

```
Age must be a number
```

The conversion fails, so the Runtime skips the rest of the `try` block and runs the `on error` block.

With valid input, the error block does not run:

```
remember age_text as "42"


try:
    convert age_text to number and call it age
    show age plus 1
on error:
    show "Age must be a number"
```

Output:

```
43
```

The `try` structure should not be used to hide unsafe behavior. Safety errors caused by blocked operations should normally remain fatal unless the Runtime explicitly allows handling them.

Invalid:

```
try:
    run shell "some system command"
on error:
```

```
show "Ignored"
```

Expected error:

Safety error: raw shell execution is not allowed in INTENTO Core.

The blocked operation is not merely a recoverable mistake. It violates the safety model.

8.14 Execution rules

INTENTO execution should follow predictable rules.

The Runtime should parse the source file before running it. If there is a syntax error, execution should not begin.

The Runtime should check names, actions, and types as early as it reasonably can. If a semantic or type error is known before execution, the program should not partially run.

Runtime errors occur during execution. When an unhandled runtime error happens, execution stops at that point.

Example:

```
show "Start"


remember divisor as 0

show 10 divided by divisor


show "End"
```

Expected output:

```
Start

Runtime error: division by zero.
```

The final line does not run.

The `stop` instruction is not an error. It is an intentional end of execution.

Example:

```
show "Start"

stop
```

show "End"

Output:

Start

The program ends after `stop`. There is no error because stopping was explicitly requested.

This distinction is important. Errors represent invalid or failed execution. `stop` represents intentional control flow.

8.15 Diagnostic information

A Runtime should give enough diagnostic information to help the programmer fix the problem.

A minimal error message should include the category and the problem.

Better error messages may also include the line number.

Example:

show "Start"

show unknown_value

Possible diagnostic:

Semantic error on line 2: unknown name `unknown_value`.

This is better than:

Error.

The Runtime should avoid vague messages. INTENTO is meant to be easy to learn, so its errors should teach.

A good diagnostic should answer three questions: what kind of error happened, where it happened, and what rule was broken.

For file and safety errors, the Runtime should also explain the blocked path or operation.

Example:

read file "/system/private.txt" and call it data

Expected error:

Safety error on line 1: path is outside the allowed workspace.

This tells the programmer that the problem is not the spelling of the instruction. The problem is the safety policy.

8.16 A complete validation and error-handling example

This example combines input, conversion, error handling, scope, and action return rules.

```
to next_year_age with age_text:
    try:
        convert age_text to number and call it age
        return age plus 1
    on error:
        return "invalid"

show "Age checker"

ask "Enter your age:" and call the answer answer

if answer is empty:
    show "Age is required"
    stop

remember result as next_year_age with answer

if result is "invalid":
    show "Age must be a number"
else:
    show "Next year you will be " plus result
```

Example interaction with valid input:

Age checker

Enter your age:

42

Next year you will be 43

Example interaction with empty input:

Age checker

Enter your age:

Age is required

Example interaction with invalid input:

Age checker

Enter your age:

abc

Age must be a number

The custom action `next_year_age` receives text, tries to convert it to a number, and returns either the calculated age or the text `"invalid"`. The main program validates empty input first, then uses the action, then decides what message to show.

This example shows why runtime rules matter. The program remains readable because each path is explicit. Empty input, invalid numeric input, and valid input are handled separately.

The Runtime does not guess. The program says what should happen.

8.17 Summary

Errors define how a language behaves when something goes wrong.

INTENTO should use clear categories: syntax errors, semantic errors, type errors, conversion errors, file errors, safety errors, runtime errors, and action errors.

Scope defines where values exist. Top-level remembered values belong to the main program memory. Custom actions have local scope. Parameters and local remembered values inside an action disappear when the action ends. Values should be passed into actions explicitly, and returned values should be used by the caller.

The `return` instruction ends an action and sends back a value. A returning action should return a value on every path when it is used as an expression.

By default, unhandled errors stop execution. The `try` and `on error` structure allows controlled recovery from ordinary recoverable errors, such as conversion failure. Safety violations should remain blocked.

The purpose of these rules is not to make INTENTO rigid. Their purpose is to make INTENTO trustworthy.

Readable code must still have exact behavior. That is what allows INTENTO to be simple for humans and reliable for machines.

Chapter 9 — Modules, Libraries, and Registered Actions

A language becomes more useful when programs can grow beyond a single file.

Small INTENTO programs can live entirely inside one `.intento` file. That is good for learning and for simple automation. But larger programs need organization. They need reusable actions, shared utilities, standard capabilities, and safe access to backend power.

This chapter introduces three related ideas: modules, libraries, and registered actions.

A module is an INTENTO file that can be reused by another INTENTO program. A library is a collection of reusable features provided by the Runtime, the standard library, or an installed package. A registered action is a controlled capability exposed to INTENTO by the Runtime, often implemented in Python or another backend layer.

The purpose is not to make INTENTO complicated. The purpose is to let INTENTO grow while keeping source code readable and execution safe.

9.1 Why modules are needed

Without modules, every program must contain all its actions in one file.

That is acceptable for a small program:

```
to greet with name:
    show "Hello " plus name


greet with "Aurora"
```

Output:

```
Hello Aurora
```

But if the same action is useful in many programs, copying it everywhere is not good design. If the action later changes, every copy must be changed too.

A module solves this problem. The action can live in one file, and other programs can use it.

For example, a file named `greetings.intento` may contain:

```
to greet with name:
    show "Hello " plus name
```

Output when the module file is loaded by itself:

(no visible output)

The file defines the action, but it does not call it. Therefore, it produces no visible output.

Another file can use that module:

```
use module "greetings.intento"


greet with "Aurora"
```

Output:

Hello Aurora

The line `use module "greetings.intento"` tells the Runtime to load the actions defined in that file. After that, the main program can call `greet`.

The important idea is simple: modules allow reusable INTENTO code to be kept in separate files.

9.2 Using a module

The basic syntax for importing a module is:

```
use module "path"
```

The path must be text. It points to another INTENTO source file.

Example module file, `formatting.intento`:

```
to show_title with title:
    show "== " plus title plus " =="
```

Output when loaded alone:

(no visible output)

Main program:

use module "formatting.intento"

show_title with "INTENTO"

show "Manual ready"

Output:

== INTENTO ==

Manual ready

The module provides the action `show_title`. The main program uses that action without redefining it.

If the module file does not exist, the Runtime should report a clear error.

Invalid:

use module "missing.intento"

show "Loaded"

Expected error:

Module error: module not found: missing.intento.

Execution should not continue, because the program may depend on actions from the missing module.

The module path should also obey the Runtime safety policy. A program should not import files from arbitrary sensitive locations unless explicitly allowed.

Invalid:

use module "/system/private_module.intento"

Expected error:

Safety error: module path is outside the allowed workspace.

Modules are useful, but they must still be controlled.

9.3 What a module should contain

A module should normally contain reusable definitions, especially custom actions.

A good module contains actions that belong together.

Example module file, `names.intento`:

```
to make_full_name with first_name and last_name:
    return first_name plus " " plus last_name


to show_name with first_name and last_name:
    remember full_name as make_full_name with first_name and last_name
    show full_name
```

Output when loaded alone:

```
(no visible output)
```

Main program:

```
use module "names.intento"

show_name with "Ada" and "Lovelace"
```

Output:

```
Ada Lovelace
```

The module defines two related actions. One returns a value, and the other shows a value. The main program does not need to know how the full name is built internally. It only calls the action.

A module should avoid doing major work immediately when imported. If importing a module unexpectedly creates files, asks questions, or changes the system, the program becomes harder to understand.

For that reason, top-level execution inside imported modules should be limited or disallowed in strict Runtime mode. A module should define capabilities; the main program should decide when to run them.

Invalid in strict module mode:

```
show "This runs during import"

to greet with name:
```

```
    show "Hello " plus name
```

Expected error when imported as a module:

Module error: top-level executable instructions are not allowed in imported module.

This rule keeps imports predictable. Loading a module should not surprise the user.

9.4 Module names and conflicts

A problem can happen when two modules define actions with the same name.

Example module `english.intento`:

```
to greet with name:
    show "Hello " plus name
```

Example module `italian.intento`:

```
to greet with name:
    show "Ciao " plus name
```

Main program:

```
use module "english.intento"
use module "italian.intento"

greet with "Aurora"
```

Expected error:

Module error: action name conflict: `greet` is defined by more than one module.

The Runtime should not guess which `greet` the programmer intended.

The solution is to use module aliases.

```
use module "english.intento" as english
use module "italian.intento" as italian

english.greet with "Aurora"
```

italian.greet with "Aurora"

Output:

Hello Aurora

Ciao Aurora

The alias makes the source of each action visible. The action `english.greet` comes from the English module. The action `italian.greet` comes from the Italian module.

Aliases are especially useful in larger programs. They prevent name conflicts and make code easier to read.

An alias should follow normal naming rules.

Invalid:

use module "english.intento" as english greetings

Expected error:

Syntax error: module alias cannot contain spaces.

Correct:

use module "english.intento" as english_greetings

english_greetings.greet with "Aurora"

Output:

Hello Aurora

9.5 Libraries

A library is a reusable collection of capabilities.

A module is usually an INTENTO file written by the programmer or project. A library is usually provided by the Runtime, the standard library, or an installed package.

The basic syntax is:

use library "name"

Example:

use library "text"

show "Library loaded"

Output:

Library loaded

The import itself produces no visible output. The visible output comes from the `show` instruction.

A library may expose registered actions.

Example:

use library "text"

remember title as "My First Project"

use action "text.slugify" with title and call the result slug

show slug

Output:

my-first-project

The library `"text"` provides the registered action `"text.slugify"`. The action receives `"My First Project"` and returns `"my-first-project"`.

If the library is not available, the Runtime should report an error.

Invalid:

use library "unknown_library"

show "Loaded"

Expected error:

Library error: unknown library `unknown_library`.

A library should not expose uncontrolled power. It should expose named, documented capabilities with clear input types, return types, and safety rules.

9.6 Registered actions

A registered action is an action known to the Runtime.

It may be implemented in Python, built into the Runtime, provided by a standard library, or loaded from a trusted package. The INTENTO program does not contain the backend implementation. It only calls the registered action through a controlled interface.

The basic syntax is:

use action "action.name" with value and call the result name

Example:

use library "text"

remember sentence as "INTENTO is readable and executable"

use action "text.count_words" with sentence and call the result count

show "Words: " plus count

Output:

Words: 5

The action `"text.count_words"` receives one text value and returns a number.

A registered action must exist before it can be used.

Invalid:

remember sentence as "Hello world"

use action "text.unknown" with sentence and call the result result

show result

Expected error:

Semantic error: unknown registered action `text.unknown`.

A registered action must also receive the correct type of value.

Invalid:

```
use library "text"

remember number as 42

use action "text.count_words" with number and call the result count

show count
```

Expected error:

Type error: action `text.count_words` expects text.

Correct version:

```
use library "text"

remember sentence as "hello world"

use action "text.count_words" with sentence and call the result count

show count
```

Output:

```
2
```

This is the main advantage of registration. The Runtime knows what the action expects and what it returns. It can check the program before or during execution instead of blindly calling backend code.

9.7 Actions with no returned value

Some registered actions may perform an operation without returning a value.

For example, a Runtime may provide a safe notification action.

use library "system"

use action "system.notify" with "Task complete"

show "Notification sent"

Possible output:

Notification sent

Expected side effect:

A notification with the text "Task complete" is shown by the system.

Because `system.notify` does not return a value, it should not be used inside `remember`.

Invalid:

use library "system"

remember result as system.notify with "Task complete"

show result

Expected error:

Type error: action `system.notify` does not return a value.

The correct form is to call it as an operation:

use library "system"

use action "system.notify" with "Task complete"

show "Done"

Possible output:

Done

The distinction is the same as with custom actions. Some actions do something. Other actions return something. A complete language must keep those behaviors separate.

9.8 Safety metadata

Every registered action should have safety metadata.

The Runtime should know whether the action is safe, requires confirmation, or is blocked in the current environment.

For example, a text action such as `text.slugify` is safe because it only transforms a value.

```
use library "text"


remember title as "INTENTO Manual"


use action "text.slugify" with title and call the result slug


show slug
```

Output:

```
intento-manual
```

A file-writing action is different. It changes the user’s environment, so it should require confirmation.

```
use library "files"


with confirmation:
    use action "files.write_text" with "notes.txt" and "Hello"


show "Write complete"
```

Expected confirmation prompt:

```
Confirm operation: files.write_text notes.txt?
```

If the user confirms, output:

Write complete

Expected file content:

Hello

If the same action is used without confirmation in a strict Runtime, the program should fail.

Invalid:

```
use library "files"


use action "files.write_text" with "notes.txt" and "Hello"


show "Write complete"
```

Expected error:

Safety error: action `files.write_text` requires confirmation.

Safety metadata makes backend power compatible with INTENTO’s identity. The source remains readable, but the Runtime still enforces limits.

9.9 Python-backed registered actions

Python can be used as a backend implementation layer for registered actions.

This does not mean that INTENTO programs should contain raw Python code. It means the Runtime can expose safe Python functions as named INTENTO actions.

Example:

```
use library "text"


remember title as "Human Readable Code"


use action "text.slugify" with title and call the result slug
```

show slug

Output:

human-readable-code

The action may be implemented in Python internally, but the INTENTO program does not need to know the implementation details.

Raw Python should not be allowed inside ordinary INTENTO programs.

Invalid:

use python code "print('hello')"

Expected error:

Safety error: raw Python code is not allowed in INTENTO programs.

This separation is essential. Python provides power. INTENTO provides controlled readable intention.

A registered Python-backed action should declare at least five things: its name, accepted parameter types, return type, safety level, and backend implementation. The programmer using INTENTO should not need to inspect the backend code to understand the action’s purpose.

A Runtime may expose this information through documentation or a command such as action inspection.

Possible inspection command:

describe action "text.slugify"

Possible output:

Action: text.slugify

Accepts: text

Returns: text

Safety: safe

Description: Converts text into a lowercase URL-safe slug.

This kind of introspection makes registered actions easier to trust.

9.10 Namespaces

A namespace is a prefix that groups names and prevents conflicts.

In this action name:

text.slugify

The namespace is:

text

The action is:

slugify

The full name is:

text.slugify

Namespaces make it possible for different libraries to use similar action names without conflict.

For example, one library may provide:

text.format

Another may provide:

date.format

These are different actions because they belong to different namespaces.

Example:

use library "text"

use library "date"

remember title as "My First Project"

remember day as "2026-06-13"

use action "text.format_title" with title and call the result formatted_title

use action "date.format_iso" with day and call the result formatted_day

show formatted_title

show formatted_day

Possible output:

The exact formatting depends on the library definitions, but the point is structural: namespaces keep names organized.

If the programmer tries to call an action without its namespace and the name is ambiguous, the Runtime should reject it.

Invalid:

use library "text"

use library "date"

use action "format" with "value" and call the result result

Expected error:

Semantic error: ambiguous action `format`. Use a namespaced action name.

Explicit names are better than hidden guessing.

9.11 Versioning

Libraries and registered actions can change over time.

A complete language should allow programs to specify the version of a library they expect.

Possible syntax:

use library "text" version "1.0"

Example:

use library "text" version "1.0"

remember title as "INTENTO Manual"

use action "text.slugify" with title and call the result slug

show slug

Output:

intento-manual

Versioning protects programs from unexpected changes. If a future `"text"` library changes how `slugify` works, a program that requests version `"1.0"` should still receive the old behavior or fail clearly if that version is unavailable.

Invalid:

use library "text" version "99.0"

Expected error:

Library error: library `text` version `99.0` is not available.

Versioning should not be required for every small script, but it becomes important for programs that must remain stable over time.

9.12 A complete modular example

This example uses a local module and a registered library action.

Module file, `reporting.intento`:

to show_report_line with label and value:

    show label plus ": " plus value

Output when loaded alone:

(no visible output)

Main program:

use module "reporting.intento"

use library "text"

show "Project report"

remember title as "My First Project"

use action "text.slugify" with title and call the result slug

```
show_report_line with "Title" and title

show_report_line with "Slug" and slug


show "Report complete"
```

Output:

```
Project report

Title: My First Project

Slug: my-first-project

Report complete
```

This program has three layers.

The main program controls the flow. The local module provides a reusable formatting action. The text library provides a registered backend action for slug creation.

The result is still readable. The program does not expose backend code. It does not duplicate formatting logic. It does not guess what `slugify` means. Each capability is named and loaded explicitly.

This is the kind of growth INTENTO should support: more power, but not less clarity.

9.13 Common errors

A common error is importing a missing module.

Invalid:

```
use module "tools.intento"


show "Ready"
```

Expected error if the file does not exist:

```
Module error: module not found: tools.intento.
```

Another common error is calling an action from a module before importing it.

Invalid:

show_title with "INTENTO"

Expected error:

Semantic error: unknown action `show_title`.

Correct:

use module "formatting.intento"

show_title with "INTENTO"

Output if `formatting.intento` defines `show_title`:

== INTENTO ==

Another common error is using an unregistered backend action.

Invalid:

use action "text.slugify" with "Hello World" and call the result slug

show slug

Expected error if the `text` library is not loaded:

Semantic error: action `text.slugify` is not available. Use the required library first.

Correct:

use library "text"

use action "text.slugify" with "Hello World" and call the result slug

show slug

Output:

hello-world

Another common error is passing the wrong number of values.

Invalid:

use library "text"

use action "text.slugify" with "Hello" and "World" and call the result slug

Expected error:

Action error: action `text.slugify` expects 1 parameter, but 2 were provided.

These errors should be precise because modules and libraries can make programs larger. The larger the program, the more important clear diagnostics become.

9.14 Summary

Modules allow INTENTO programs to reuse actions from other `.intento` files.

Libraries provide reusable capabilities from the Runtime, the standard library, or installed packages.

Registered actions expose controlled backend power to INTENTO. They may be implemented in Python or another system, but they must be named, typed, documented, and safety-checked.

Namespaces prevent conflicts. Aliases help when multiple modules expose similar names. Versioning allows programs to depend on stable library behavior.

The most important rule is that extension must not destroy readability. INTENTO should grow through explicit, named capabilities, not through hidden backend code or uncontrolled execution.

A complete INTENTO program should make its dependencies visible:

use module "reporting.intento"

use library "text"

Output:

(no visible output)

These lines tell the reader what the program depends on. They also tell the Runtime what capabilities must be loaded before execution.

This is how INTENTO can become larger without becoming obscure: every added power must remain readable, registered, and controlled.

Chapter 10 — Standard Library and Real Programs

A language becomes truly useful when it provides common tools that programmers can rely on.

The previous chapter introduced modules, libraries, namespaces, and registered actions. This chapter completes that idea by defining the role of the INTENTO Standard Library and showing how real programs can be written by combining the language core with safe, documented capabilities.

The Standard Library is not meant to make INTENTO heavy. It is meant to prevent every programmer from solving the same basic problems again and again. Text processing, list operations, file helpers, dates, simple data formats, logs, and safe integrations should be available through clear, registered actions.

The central rule remains the same: every capability must be readable in the INTENTO source and controlled by the Runtime.

10.1 What the Standard Library is

The Standard Library is the official collection of reusable capabilities provided with the INTENTO Runtime.

It is not raw backend code exposed to the user. It is a set of named libraries and registered actions that follow INTENTO rules: clear names, declared input types, declared return types, predictable behavior, and safety metadata.

A program can load a standard library with `use library`.

Example:

use library "text"

show "Text library loaded"

Output:

Text library loaded

The line `use library "text"` loads the library. It does not display anything by itself. The visible output comes from `show`.

If the library does not exist, the Runtime should report a clear error.

use library "unknown"

show "Ready"

Expected error:

Library error: unknown library `unknown`.

A standard library should be stable. If a program uses an official action such as `text.count_words`, the programmer should be able to trust that the action has documented behavior.

10.2 Standard library design

A good INTENTO standard library should be organized by namespaces.

Examples:

text

number

list

file

date

json

csv

log

system

Each namespace groups related actions.

For example, the `text` library may provide actions such as:

text.slugify

text.count_words

text.uppercase

text.lowercase

text.trim

The `list` library may provide:

list.count

list.join

list.contains

The `file` library may provide safe file helpers, while still respecting confirmation and workspace rules.

The important point is not the exact number of actions in the first Runtime. The important point is the design rule: standard actions must be named clearly and behave predictably.

A programmer should be able to read this:

use action "text.count_words" with sentence and call the result count

and understand the intention without seeing the backend implementation.

10.3 Text tools

Text processing is one of the most important parts of a practical language.

A standard `text` library may provide actions for counting words, changing case, trimming spaces, and creating safe slugs.

Example:

use library "text"

remember title as " Human Readable Code "

use action "text.trim" with title and call the result clean_title

use action "text.slugify" with clean_title and call the result slug

use action "text.count_words" with clean_title and call the result words

show "Title: " plus clean_title

show "Slug: " plus slug

show "Words: " plus words

Output:

Title: Human Readable Code

Slug: human-readable-code

Words: 3

This example shows three standard text actions.

The action `text.trim` removes leading and trailing spaces. The action `text.slugify` creates a lowercase, path-friendly version of the title. The action `text.count_words` returns the number of words.

If the wrong type is passed, the Runtime should reject the call.

use library "text"

remember value as 42

use action "text.count_words" with value and call the result words

show words

Expected error:

Type error: action `text.count_words` expects text.

The error is important. The Runtime should not convert the number to text secretly unless the action explicitly allows that behavior.

10.4 Number and list tools

The core language already supports basic arithmetic and lists. The Standard Library can add practical helpers.

A `list` library may count items, join items, or check whether a list contains a value.

Example:

use library "list"

remember skills as ["Linux", "Python", "INTENTO"]

use action "list.count" with skills and call the result skill_count

use action "list.join" with skills and ", " and call the result skill_text

show "Skills: " plus skill_text

show "Total skills: " plus skill_count

Output:

Skills: Linux, Python, INTENTO

Total skills: 3

The list itself remains a list. The joined value is a new text value created for display.

If an action expects a list and receives text, the Runtime should report a type error.

use library "list"

remember skills as "Linux, Python, INTENTO"

use action "list.count" with skills and call the result skill_count

show skill_count

Expected error:

Type error: action `list.count` expects list.

This is strict, but useful. A text that contains commas is not the same as a list. If the programmer wants to split text into a list, that should be done by a clear action such as `text.split`, not by hidden guessing.

10.5 Date tools

Dates are common in real programs, but date handling can become complex. For this reason, date operations should live in the Standard Library instead of being improvised in the core syntax.

A `date` library may parse, format, and compare dates.

Example:

use library "date"

remember raw_date as "2026-06-13"

use action "date.format" with raw_date and "long" and call the result readable_date

show "Date: " plus readable_date

Possible output:

Date: June 13, 2026

The exact formats supported by the Runtime should be documented. If a date is invalid, the Runtime should report a clear error.

use library "date"

remember raw_date as "not-a-date"

use action "date.format" with raw_date and "long" and call the result readable_date

show readable_date

Expected error:

Date error: cannot parse date `not-a-date`.

Dates should not be guessed casually. Different countries write dates differently, and ambiguous date parsing can create serious mistakes. INTENTO should prefer explicit formats.

10.6 JSON and CSV tools

Many real programs exchange data using structured formats such as JSON and CSV.

INTENTO does not need to make JSON or CSV part of the core syntax. Instead, the Standard Library can expose safe actions for reading, parsing, and writing these formats.

Example with JSON text:

use library "json"

remember data as "{\"name\":\"Aurora\",\"language\":\"INTENTO\"}"

use action "json.get" with data and "name" and call the result name

use action "json.get" with data and "language" and call the result language

show "Name: " plus name

show "Language: " plus language

Output:

Name: Aurora

Language: INTENTO

The action `json.get` receives JSON text and a key, then returns the value associated with that key.

If the JSON is invalid, the Runtime should report an error.

use library "json"

remember data as "{bad json}"

use action "json.get" with data and "name" and call the result name

show name

Expected error:

JSON error: invalid JSON text.

CSV can follow the same principle. A CSV action should parse data according to documented rules instead of relying on vague text splitting.

Example:

use library "csv"

remember rows as "name,role\nAda,mathematician\nGrace,computer scientist"

use action "csv.count_rows" with rows and call the result row_count

show "Rows: " plus row_count

Output:

Rows: 2

The header line is not counted as a data row in this example. That rule must be documented by the action. Standard library behavior should always be explicit.

10.7 Logging

Real programs should explain what they did.

Simple output with `show` is useful for the user, but logs are useful for diagnosis. A log records events that happened during execution.

A `log` library may provide safe logging actions.

Example:

use library "log"

show "Starting task"

use action "log.info" with "Task started"

use action "log.info" with "Task finished"

show "Task finished"

Output:

Starting task

Task finished

Expected log entries:

INFO Task started

INFO Task finished

The log entries may be written to the Runtime log, not necessarily to the visible program output.

Logging should be safe by default. It should not write sensitive data unless the program explicitly provides it, and the Runtime should decide where logs are stored.

A logging action that writes to a user file may require confirmation, depending on the Runtime configuration.

10.8 Controlled integrations

Some INTENTO programs may need controlled access to external systems: sending a notification, calling a local service, querying an API, or interacting with a database.

These features should not be part of the unrestricted core language. They should be exposed as registered actions with safety metadata.

Example:

use library "system"

use action "system.notify" with "INTENTO task complete"

show "Notification requested"

Possible output:

Notification requested

Expected side effect:

A system notification is shown with the text "INTENTO task complete".

A network action should be stricter because it communicates outside the local machine.

Example:

use library "network"

with confirmation:

    use action "network.fetch_text" with "https://example.com/status" and call the result status

show status

Possible confirmation prompt:

Confirm operation: network.fetch_text https://example.com/status?

Possible output if confirmed:

OK

If the Runtime does not allow network access, the action should be blocked.

Expected error in a restricted Runtime:

Safety error: network access is not allowed in this Runtime configuration.

This keeps INTENTO honest. A program should not silently contact external systems.

10.9 Real program: project initializer

This complete example creates a small project structure. It uses input, validation, text tools, file operations, confirmation, and readable output.

```
use library "text"


show "Project initializer"


ask "Project title?" and call the answer title


if title is empty:
    show "Project title is required"
    stop


use action "text.slugify" with title and call the result slug


remember folder as "Projects/" plus slug

remember readme as folder plus "/README.txt"

remember readme_text as "# " plus title
```

with confirmation:

    create folder folder

    create file readme with readme_text

show "Project created: " plus folder

Example interaction:

Project initializer

Project title?

Human Terminal

Confirm operation: create folder Projects/human-terminal?

Confirm operation: create file Projects/human-terminal/README.txt?

Project created: Projects/human-terminal

Expected filesystem result:

Projects/human-terminal/

Projects/human-terminal/README.txt

Expected README content:

Human Terminal

This program shows why INTENTO can be both readable and practical. The source does not expose Python code or shell commands. It expresses the intention clearly: ask for a title, make a safe folder name, build paths, ask for confirmation, create the project, and show the result.

The Runtime handles the technical details and enforces the safety rules.

10.10 Real program: note analyzer

This example reads a note file, counts its words, and prints a small report.

use library "text"

```
show "Note analyzer"

read file "notes.txt" and call it notes

use action "text.count_words" with notes and call the result words

show "File: notes.txt"
show "Words: " plus words
```

Expected output if `notes.txt` contains `INTENTO is readable code`:

```
Note analyzer
File: notes.txt
Words: 4
```

If the file does not exist, the Runtime should report an error.

```
use library "text"

show "Note analyzer"

read file "missing.txt" and call it notes

use action "text.count_words" with notes and call the result words

show "Words: " plus words
```

Expected error:

```
File error: file not found: missing.txt.
```

The program stops at the file error. It does not continue to count words because the value `notes` was never created.

A more advanced version could use `try` and `on error` to handle the missing file gracefully.

```
use library "text"

show "Note analyzer"

try:
    read file "missing.txt" and call it notes
    use action "text.count_words" with notes and call the result words
    show "Words: " plus words
on error:
    show "Could not read the note file"
```

Output:

```
Note analyzer
Could not read the note file
```

This version is better for user-facing programs because it replaces a technical failure with a clear message.

10.11 Real program: skill report

This example uses lists, a standard library action, a custom action, and a loop.

```
use library "list"

show "Skill report"

remember skills as ["Linux", "Python", "INTENTO"]

use action "list.count" with skills and call the result total

to show_skill with skill:
```

```
    show "- " plus skill


show "Total skills: " plus total


for each skill in skills:
    show_skill with skill


show "Report complete"
```

Output:

```
Skill report
Total skills: 3
- Linux
- Python
- INTENTO
Report complete
```

The program remains readable because each part has a clear role.

The list stores the data. The standard library counts the items. The custom action formats one skill. The loop applies the action to every skill.

This is the kind of structure INTENTO should encourage. The code is not compressed into clever tricks. It is organized into named intentions.

10.12 Distribution and execution

A complete INTENTO program may depend on modules, libraries, registered actions, files, and Runtime permissions.

For this reason, distribution should be explicit.

A project may contain:

```
main.intento
modules/
```

data/

README.txt

intento.project

The file `main.intento` contains the main program. The `modules` folder contains reusable `.intento` modules. The `data` folder contains local data files. The `README.txt` explains how to run the program. The optional `intento.project` file may declare required libraries, permissions, and Runtime version.

A simple project declaration may say:

runtime: INTENTO 1.0

libraries: text, list, file

permissions: read project files, write project files with confirmation

This is not executable INTENTO code. It is project metadata. Its purpose is to tell the Runtime and the user what the program needs.

A Runtime should refuse to run a program if required capabilities are missing.

Expected error:

Project error: required library `text` is not available.

This is better than failing later in the middle of execution.

10.13 Standard library safety

The Standard Library must follow the same safety model as the rest of INTENTO.

Safe actions may run without confirmation. Text transformation, word counting, list counting, and date formatting are examples of safe actions.

Actions that write files, modify the system, access the network, send notifications, or call external services may require confirmation or explicit Runtime permission.

Blocked actions should not run at all in the current Runtime configuration.

Example:

use library "network"

use action "network.fetch_text" with "https://example.com" and call the result page

show page

Expected error in a Runtime without network permission:

Safety error: network access is not allowed in this Runtime configuration.

The same program may be valid in a Runtime configured for controlled network access, especially if placed inside `with confirmation`.

The source code alone is not the whole story. Runtime configuration matters. INTENTO programs must be readable, but execution must also respect the environment where they run.

10.14 What makes a real INTENTO program

A real INTENTO program should have four qualities.

It should be readable. A human should understand the intention of the program by reading the source.

It should be structured. Repeated behavior should use loops or actions. Shared behavior should use modules or libraries.

It should be explicit. Type conversion, file access, network access, and backend actions should be visible in the source.

It should be safe. Sensitive operations should require confirmation or be blocked by policy.

This example is readable but incomplete:

ask "Project title?" and call the answer title

show "Creating " plus title

Example interaction:

Project title?

INTENTO

Creating INTENTO

It communicates, but it does not validate input or create anything.

This version is closer to a real program:

use library "text"

ask "Project title?" and call the answer title

```
if title is empty:
    show "Project title is required"
    stop

use action "text.slugify" with title and call the result slug

remember folder as "Projects/" plus slug

with confirmation:
    create folder folder

show "Created: " plus folder
```

Example interaction:

```
Project title?
INTENTO Manual
Confirm operation: create folder Projects/intento-manual?
Created: Projects/intento-manual
```

Expected filesystem result:

```
Projects/intento-manual/
```

The second program checks input, uses a standard text action, builds a path, asks for confirmation, changes the filesystem safely, and shows the result. It is still easy to read.

That is the target shape of INTENTO.

10.15 Summary

The INTENTO Standard Library gives programs reliable tools without making the core language heavy.

Libraries such as `text`, `list`, `date`, `json`, `csv`, `log`, `file`, `system`, and `network` should expose registered actions with clear names, declared types, return values, and safety metadata.

Real programs combine the language core with these capabilities. They ask for input, validate values, convert types explicitly, use actions and loops to stay organized, access files with safety rules, and rely on registered backend power only through controlled interfaces.

The Standard Library should make INTENTO practical, but it should not weaken the language's identity.

INTENTO remains the readable layer.

The Runtime remains the controlled execution layer.

The Standard Library is the bridge between them.

A complete INTENTO language is not defined by how many features it has. It is defined by whether those features remain understandable, predictable, and safe when used together.

INTENTO Language Reference

Official Draft 0.1

Italian name. English syntax. Human-readable code. Real execution.

This document defines the official commands, structures, values, and runtime rules of INTENTO Draft 0.1.

The manual explains the language.
The Language Reference specifies it.

A command belongs in INTENTO only if its syntax, meaning, expected behavior, and safety level are clear.

---

1. Safety Levels

Every INTENTO instruction belongs to a safety level.

Safe

Safe instructions do not modify the user's files, folders, system, network, or external services.

Examples:

show

remember

if

else

repeat

for each

to

return

convert

Safe instructions may run without confirmation.

Confirmation Required

These instructions can modify the user's environment and should normally run inside `with confirmation`.

Examples:

create folder

create file

append to file

registered actions that write files

registered actions that access network

registered actions that modify external systems

Blocked

Blocked operations are not allowed in INTENTO Draft 0.1.

Examples:

raw shell execution

raw Python code inside INTENTO source

unrestricted deletion

unrestricted overwrite

paths outside the allowed workspace

unregistered backend actions

Confirmation does not make blocked operations valid.

---

2. Values and Types

INTENTO Draft 0.1 supports these fundamental value types:

text

number

boolean

empty

list

Text

Text must be written between quotation marks.

show "Hello"

Output:

Hello

Safety level: safe.

Number

Numbers are written without quotation marks.

show 42

Output:

42

Safety level: safe.

Boolean

Booleans are:

true

false

Example:

show true

show false

Output:

true

false

Safety level: safe.

Empty

The value `empty` represents absence of content.

show empty

Output:

(empty line)

Safety level: safe.

List

A list contains multiple values in order.

show ["Linux", "Python", "INTENTO"]

Output:

["Linux", "Python", "INTENTO"]

Safety level: safe.

---

3. Names

Names are used for remembered values, parameters, actions, modules, and aliases.

Recommended name rules:

starts with a lowercase letter

may contain lowercase letters, numbers, and underscores

must not contain spaces

must not be a keyword

Valid names:

name

project_name

total_price

item2

Invalid:

remember project name as "INTENTO"

Expected error:

Syntax error: names cannot contain spaces.

Safety level: safe.

---

4. Program Structure

An INTENTO program is a `.intento` text file.

The Runtime reads the file from top to bottom.

Basic rules:

one instruction per line

blank lines are allowed

comments start with #

blocks end with :

block contents are indented

Example:

First program

show "Hello"

Output:

Hello

Safety level: safe.

---

5. Comments

Comments start with #.

Syntax:

comment text

Example:

This program shows a greeting

show "Hello"

Output:

Hello

Safety level: safe.

---

6. `show`

Displays a value.

Syntax:

show value

Example:

show "Hello INTENTO"

Output:

Hello INTENTO

Example with memory:

remember name as "Aurora"

show "Hello " plus name

Output:

Hello Aurora

Invalid:

show

Expected error:

Syntax error: `show` needs a value.

Safety level: safe.

---

7. `remember`

Stores a value in program memory.

Syntax:

remember name as value

Example:

remember language as "INTENTO"

show language

Output:

INTENTO

A remembered value must exist before it is used.

Invalid:

show language

Expected error:

Semantic error: unknown name `language`.

Safety level: safe.

---

8. Expressions

An expression is code that produces a value.

Examples:

show 10 plus 5

show "Hello " plus "world"

Output:

15

Hello world

Safety level: safe, unless the expression calls an action with a different safety level.

---

9. Operators

Canonical INTENTO operators are literary word operators.

plus

minus

times

divided by

`plus`

Adds numbers or joins text.

show 10 plus 5

show "Hello " plus "world"

Output:

15

Hello world

`minus`

Subtracts numbers.

show 10 minus 5

Output:

5

`times`

Multiplies numbers.

show 10 times 5

Output:

50

`divided by`

Divides numbers.

show 10 divided by 5

Output:

2

Invalid:

show "10" minus 5

Expected error:

Type error: `minus` requires numbers.

Invalid:

show 10 divided by 0

Expected error:

Runtime error: division by zero.

Safety level: safe.

Symbolic aliases such as `+`, `-`, `*`, and `/` may be supported by Runtime versions as compact equivalents, but the canonical documented form remains word-based.

---

10. Parentheses

Parentheses group expressions and control evaluation order.

Syntax:

(expression)

Example:

show (2 plus 3) times 4

Output:

20

Without parentheses:

show 2 plus 3 times 4

Output:

14

Invalid:

show (2 plus 3

Expected error:

Syntax error: missing closing parenthesis.

Safety level: safe.

---

11. `ask`

Asks the user for input and stores the answer.

Syntax:

ask "Question?" and call the answer name

Example:

ask "What is your name?" and call the answer name

show "Hello " plus name

Example interaction:

What is your name?

Aurora

Hello Aurora

Invalid:

ask What is your name? and call the answer name

Expected error:

Syntax error: question text must be written between quotation marks.

Safety level: safe.

Input should be treated as text until explicitly converted.

---

12. `if`

Runs a block only if a condition is true.

Syntax:

```
if condition:
    instruction
```

Example:

```
remember name as empty

if name is empty:
    show "Name is required"
```

Output:

Name is required

Invalid:

```
if name is empty
    show "Name is required"
```

Expected error:

Syntax error: expected `:` after condition.

Safety level: safe.

---

13. `else`

Runs an alternative block when the preceding `if` condition is false.

Syntax:

```
if condition:
    instruction
else:
    instruction
```

Example:

```
remember name as "Aurora"


if name is empty:
    show "Name is required"
else:
    show "Hello " plus name
```

Output:

```
Hello Aurora
```

Invalid:

```
else:
    show "No condition"
```

Expected error:

```
Syntax error: `else` must follow an `if` block.
```

Safety level: safe.

---

14. Comparisons

Comparisons are used in conditions.

Official comparison forms:

is

is not

is empty

is greater than

is less than

is at least

is at most

Examples:

remember age as 20

```
if age is at least 18:
    show "Access allowed"
else:
    show "Access denied"
```

Output:

Access allowed

Example with text:

remember language as "INTENTO"

```
if language is "INTENTO":
    show "Correct"
else:
    show "Wrong"
```

Output:

Correct

Invalid:

remember age as "20"

if age is at least 18:

    show "Access allowed"

Expected error:

Type error: cannot compare text with number using `is at least`.

Safety level: safe.

---

15. `stop`

Ends program execution immediately.

Syntax:

stop

Example:

show "Start"

stop

show "End"

Output:

Start

Invalid:

stop "now"

Expected error:

Syntax error: `stop` does not take a value.

Safety level: safe.

---

16. `repeat`

Runs a block a fixed number of times.

Syntax:

```
repeat number times:
    instruction
```

Example:

```
repeat 3 times:
    show "Hello"
```

Output:

```
Hello
Hello
Hello
```

Example with memory:

```
remember counter as 0

repeat 3 times:
    remember counter as counter plus 1
    show counter
```

Output:

```
1
2
3
```

Invalid:

```
repeat "3" times:
    show "Hello"
```

Expected error:

Type error: repeat count must be a number.

Safety level: safe.

---

17. Lists and `for each`

Processes each item in a list.

Syntax:

for each item in list:

    instruction

Example:

remember skills as ["Linux", "Python", "INTENTO"]

for each skill in skills:

    show skill

Output:

Linux

Python

INTENTO

Invalid:

remember skill as "Linux"

for each item in skill:

    show item

Expected error:

Type error: `for each` requires a list after `in`.

Safety level: safe.

---

18. Custom Actions with `to`

Defines a reusable action.

Syntax without parameters:

```
to action_name:
    instruction
```

Example:

```
to greet:
    show "Hello"

greet
```

Output:

```
Hello
```

Defining an action does not run it.

```
to greet:
    show "Hello"

show "Ready"
```

Output:

```
Ready
```

Safety level: safe, unless the action body contains confirmation-required or blocked instructions.

---

19. Action Parameters

Defines an action that receives values.

Syntax:

```
to action_name with parameter:
    instruction

action_name with value
```

Example:

```
to greet with name:
    show "Hello " plus name

greet with "Aurora"
```

Output:

```
Hello Aurora
```

Multiple parameters:

```
to show_profile with name and role:
    show name plus " — " plus role

show_profile with "Ada" and "mathematician"
```

Output:

```
Ada — mathematician
```

Invalid:

```
to greet with name:
    show "Hello " plus name

greet
```

Expected error:

```
Action error: action `greet` needs 1 parameter.
```

Safety level: depends on action body.

---

20. `return`

Returns a value from a custom action.

Syntax:

return value

Example:

```
to make_greeting with name:
    return "Hello " plus name

remember message as make_greeting with "Aurora"
show message
```

Output:

```
Hello Aurora
```

Invalid:

```
return "Hello"
```

Expected error:

```
Syntax error: `return` can only be used inside a custom action.
```

Invalid:

```
to greet with name:
    show "Hello " plus name

remember message as greet with "Aurora"
```

Expected error:

```
Action error: action `greet` does not return a value.
```

Safety level: safe.

---

21. `convert`

Converts a value to another type explicitly.

Syntax:

```
convert value to type and call it name
```

Supported target types:

text

number

boolean

Example:

remember age_text as "42"

convert age_text to number and call it age

show age plus 1

Output:

43

Invalid:

remember age_text as "forty-two"

convert age_text to number and call it age

Expected error:

Conversion error: cannot convert `forty-two` to number.

Boolean conversion:

remember answer as "true"

convert answer to boolean and call it active

show active

Output:

true

Invalid:

remember answer as "yes"

convert answer to boolean and call it active

Expected error:

Conversion error: cannot convert `yes` to boolean.

Safety level: safe.

---

22. `try` / `on error`

Handles recoverable errors.

Syntax:

try:

    instruction

on error:

    instruction

Example:

remember age_text as "abc"

try:

    convert age_text to number and call it age

    show age plus 1

on error:

    show "Age must be a number"

Output:

Age must be a number

Example without error:

```
remember age_text as "42"


try:
    convert age_text to number and call it age
    show age plus 1
on error:
    show "Age must be a number"
```

Output:

```
43
```

Safety errors caused by blocked operations should not normally be recoverable.

Invalid:

```
try:
    run shell "some command"
on error:
    show "Ignored"
```

Expected error:

```
Safety error: raw shell execution is not allowed in INTENTO.
```

Safety level: safe for ordinary recoverable errors.

---

23. `with confirmation`

Requires user confirmation before sensitive operations.

Syntax:

```
with confirmation:
    instruction
```

Example:

```
with confirmation:
```

```
create folder "Projects/Test"
```

```
show "Done"
```

Expected confirmation prompt:

Confirm operation: create folder Projects/Test?

If confirmed, output:

Done

If refused, possible output:

Operation cancelled by user.

Safety level: confirmation required.

---

24. `create folder`

Creates a folder.

Syntax:

create folder path

Example:

with confirmation:

```
create folder "Projects/INTENTO"
```

```
show "Folder ready"
```

Expected confirmation prompt:

Confirm operation: create folder Projects/INTENTO?

If confirmed, output:

Folder ready

Expected filesystem result:

Projects/INTENTO/

Invalid:

with confirmation:

    create folder "/system/private"

Expected error:

Safety error: path is outside the allowed workspace.

Safety level: confirmation required.

---

25. `create file`

Creates a file with content.

Syntax:

create file path with content

Example:

with confirmation:

    create file "notes.txt" with "Hello"

show "File created"

Expected confirmation prompt:

Confirm operation: create file notes.txt?

If confirmed, output:

File created

Expected file content:

Hello

If the file already exists:

Safety error: file already exists. Use an explicit overwrite operation if replacement is intended.

Safety level: confirmation required.

---

26. `read file`

Reads a file and stores its content.

Syntax:

read file path and call it name

Example:

read file "notes.txt" and call it notes

show notes

Expected output if `notes.txt` contains `Hello`:

Hello

Invalid:

read file "missing.txt" and call it notes

show notes

Expected error:

File error: file not found: missing.txt.

Safety level: safe if path is inside allowed workspace; otherwise blocked.

---

27. `append to file`

Adds content to the end of an existing file.

Syntax:

append to file path with content

Example:

with confirmation:

```
    append to file "log.txt" with "Program started"


show "Log updated"
```

Expected confirmation prompt:

Confirm operation: append to file log.txt?

If confirmed, output:

Log updated

Expected file effect:

Program started

Invalid if file does not exist:

with confirmation:

```
append to file "missing-log.txt" with "Entry"
```

Expected error:

File error: cannot append because file does not exist: missing-log.txt.

Safety level: confirmation required.

---

28. `delete file`

Not available in INTENTO Draft 0.1.

Invalid:

delete file "notes.txt"

Expected error:

Safety error: deletion is not available in INTENTO Draft 0.1.

Safety level: blocked.

---

29. `overwrite file`

Reserved for future versions.

Possible future syntax:

overwrite file path with content

In Draft 0.1, this should not be available by default.

Invalid:

overwrite file "notes.txt" with "New text"

Expected error:

Safety error: overwrite is not available in INTENTO Draft 0.1.

Safety level: blocked or confirmation required in future versions.

---

30. `use module`

Imports reusable INTENTO code from another `.intento` file.

Syntax:

use module "path"

Example:

use module "formatting.intento"

show_title with "INTENTO"

Expected output if `formatting.intento` defines `show_title`:

== INTENTO ==

With alias:

use module "formatting.intento" as formatting

formatting.show_title with "INTENTO"

Expected output:

== INTENTO ==

Invalid:

use module "missing.intento"

Expected error:

Module error: module not found: missing.intento.

Safety level: safe if module path is inside allowed workspace.

---

31. `use library`

Loads a registered library.

Syntax:

use library "name"

Optional version syntax:

use library "name" version "1.0"

Example:

use library "text"

show "Library loaded"

Output:

Library loaded

Invalid:

use library "unknown"

Expected error:

Library error: unknown library `unknown`.

Safety level: depends on library; loading itself is safe.

---

32. `use action`

Calls a registered action from a library or Runtime.

Syntax when the action returns a value:

use action "namespace.action" with value and call the result name

Syntax when the action does not return a value:

use action "namespace.action" with value

Example with returned value:

```
use library "text"

remember title as "Human Readable Code"

use action "text.slugify" with title and call the result slug

show slug
```

Output:

```
human-readable-code
```

Example without returned value:

```
use library "system"

use action "system.notify" with "Task complete"

show "Done"
```

Possible output:

```
Done
```

Invalid unknown action:

```
use action "text.unknown" with "Hello" and call the result value
```

Expected error:

```
Semantic error: unknown registered action `text.unknown`.
```

Safety level: defined by action metadata.

Official form: `use action`.
Older wording such as `use python action` should be treated as descriptive, not the preferred official syntax.

---

33. `describe action`

Displays metadata for a registered action.

Syntax:

describe action "namespace.action"

Example:

use library "text"

describe action "text.slugify"

Possible output:

Action: text.slugify

Accepts: text

Returns: text

Safety: safe

Description: Converts text into a lowercase URL-safe slug.

Invalid:

describe action "text.unknown"

Expected error:

Semantic error: unknown registered action `text.unknown`.

Safety level: safe.

---

34. Modules and Namespaces

A namespace groups actions and prevents name conflicts.

Example:

text.slugify

date.format

list.count

The part before the dot is the namespace.
The part after the dot is the action name.

If two modules define the same action name, aliases should be used.

Example:

use module "english.intento" as english

use module "italian.intento" as italian

english.greet with "Aurora"

italian.greet with "Aurora"

Output:

Hello Aurora

Ciao Aurora

Safety level: safe.

---

35. Standard Library Namespaces

Recommended official namespaces:

text

number

list

file

date

json

csv

log

system

network

Each action must declare:

name

accepted parameter types

return type

safety level

description

Example:

use library "list"

remember skills as ["Linux", "Python", "INTENTO"]

use action "list.count" with skills and call the result total

show total

Output:

3

Safety level: depends on action.

---

36. Blocked Raw Execution

Raw shell execution is blocked.

Invalid:

run shell "ls"

Expected error:

Safety error: raw shell execution is not allowed in INTENTO.

Raw Python code is blocked.

Invalid:

use python code "print('hello')"

Expected error:

Safety error: raw Python code is not allowed in INTENTO programs.

Safety level: blocked.

---

37. Error Categories

Official error categories:

Syntax error

Semantic error

Type error

Conversion error

File error

Module error

Library error

Action error

Runtime error

Safety error

Project error

Example:

show unknown_value

Expected error:

Semantic error: unknown name `unknown_value`.

Example with line number:

Semantic error on line 1: unknown name `unknown_value`.

The Runtime should prefer clear diagnostic messages.

---

38. Execution Rules

The Runtime should follow these rules.

First, parse the source file. If syntax is invalid, execution does not begin.

Then check names, actions, types, and safety rules as early as possible.

During execution, run instructions from top to bottom unless flow is changed by:

if

else

repeat

for each

action calls

return

stop

errors

Unhandled errors stop execution.

Example:

show "Start"

show 10 divided by 0

show "End"

Expected output:

Start

Runtime error: division by zero.

The final instruction does not run.

---

39. Scope Rules

Top-level remembered values belong to main program memory.

Custom actions have local scope.

Example:

```
to make_message with name:
    remember message as "Hello " plus name
    return message

remember result as make_message with "Aurora"
show result
```

Output:

```
Hello Aurora
```

The local value `message` is not available outside the action.

Invalid:

```
to make_message with name:
    remember message as "Hello " plus name
    return message

show message
```

Expected error:

```
Semantic error: unknown name `message`.
```

Safety level: safe.

---

40. Project Metadata

A larger INTENTO project may include a metadata file.

Possible file:

```
intento.project
```

Possible content:

```
runtime: INTENTO 1.0
```

libraries: text, list, file

permissions: read project files, write project files with confirmation

This is not executable INTENTO code. It describes the project requirements.

If a required library is missing:

Project error: required library `text` is not available.

Safety level: safe.

---

41. Complete Example

```
use library "text"

show "Project initializer"

ask "Project title?" and call the answer title

if title is empty:
    show "Project title is required"
    stop

use action "text.slugify" with title and call the result slug

remember folder as "Projects/" plus slug
remember readme as folder plus "/README.txt"
remember readme_text as "# " plus title

with confirmation:
    create folder folder
```

create file readme with readme_text

show "Project created: " plus folder

Example interaction:

Project initializer

Project title?

Human Terminal

Confirm operation: create folder Projects/human-terminal?

Confirm operation: create file Projects/human-terminal/README.txt?

Project created: Projects/human-terminal

Expected filesystem result:

Projects/human-terminal/

Projects/human-terminal/README.txt

Expected README content:

Human Terminal

This example uses:

use library

show

ask

if

stop

use action

remember

plus

with confirmation

create folder

create file

It represents the intended style of INTENTO: readable source, explicit dependencies, validation, safe file operations, and controlled backend power.

---

42. Reserved for Future Versions

The following capabilities are not official in Draft 0.1, but may be introduced later:

while

repeat until

and

or

not

dictionary/object values

records

classes

async tasks

package manager

database actions

HTTP actions

testing syntax

debug commands

overwrite file

delete file

Future features must follow the same rule:

readable syntax

precise parsing

clear safety level

documented behavior

expected output or error

INTENTO should grow carefully.

---

43. Final Rule

INTENTO is not free natural language.

INTENTO is not raw Python.

INTENTO is not shell scripting with prettier words.

INTENTO is a human-readable programming language with controlled syntax and real execution.

Every official command must answer five questions:

What is the syntax?

What does it mean?

What value does it produce, if any?

What output or error should be expected?

What safety level does it have?

If a command cannot answer these questions, it does not belong in the official language yet.

Remaining Chapters Index

Chapter 11 — Official Language Reference

This chapter defines the official INTENTO language commands in a compact and precise form. Each command is described by its syntax, meaning, expected behavior, example output or expected error, and safety level.

It includes the official reference for instructions such as `show`, `remember`, `ask`, `if`, `else`, `repeat`, `for each`, `to`, `return`, `convert`, `try`, `on error`, `with confirmation`, file commands, modules, libraries, and registered actions.

Chapter 12 — Standard Library Reference

This chapter defines the official INTENTO Standard Library. It describes the main library namespaces and their registered actions, including text, number, list, file, date, JSON, CSV, log, system, and network utilities.

Each library action should specify its name, accepted input types, returned value, safety level, and a compact example with expected output or expected error.

Chapter 13 — Runtime, Project Format, and Implementation Rules

This chapter defines how INTENTO programs are executed in practice. It describes the Runtime, workspace rules, permissions, project structure, project metadata, execution logs, CLI behavior, module loading, registered Python-backed actions, and implementation requirements.

Its purpose is to make INTENTO not only a language specification, but also a practical system that can be implemented, tested, distributed, and executed safely.

Chapter 11 — Official Language Reference

This chapter defines the official language surface of INTENTO.

The previous chapters explained the language step by step. This chapter has a different purpose: it collects the official commands, structures, values, and runtime expectations in one compact reference.

A command belongs to INTENTO only if its syntax, meaning, behavior, and safety level are clear. INTENTO must remain human-readable, but it must not become vague. Every official instruction must be understandable for the human reader and precise enough for the Runtime.

This chapter is not a replacement for the manual. It is the formal reference that makes the manual executable.

11.1 Program files and structure

An INTENTO program is written in a plain text file with the `.intento` extension.

A program is executed from top to bottom unless the flow is changed by a condition, a loop, a custom action, `return`, `stop`, or an error.

The basic structural rules are simple. Each instruction is written on its own line. Blank lines are allowed. Comments begin with #. Blocks begin after a line ending with `:`, and the instructions inside the block are indented.

Example:

```
# First INTENTO program


show "Hello from INTENTO"
```

Output:

Hello from INTENTO

The comment is ignored by the Runtime. The `show` instruction runs and displays the text.

Invalid structure should be rejected before execution.

```
if name is empty
   show "Name is required"
```

Expected error:

Syntax error: expected `:` after condition.

The Runtime should not guess where a block begins. The colon is part of the language structure.

Safety level: safe.

11.2 Values and types

INTENTO supports a small set of fundamental value types: text, number, boolean, empty, and list.

Text is written between quotation marks. Numbers are written without quotation marks. Booleans are written as `true` or `false`. The value `empty` represents absence of content. Lists are written between square brackets.

Example:

```
show "INTENTO"
show 42
show true
show empty
show ["Linux", "Python", "INTENTO"]
```

Output:

```
INTENTO
42
true

["Linux", "Python", "INTENTO"]
```

Types matter because the Runtime must know how to handle each value. The text `"42"` is not the same as the number `42`.

Example:

show "42" plus "8"

show 42 plus 8

Output:

428

50

The first expression joins text. The second expression adds numbers.

Invalid type use should be rejected clearly.

show "42" minus 8

Expected error:

Type error: `minus` requires numbers.

Safety level: safe.

11.3 Names

Names are used for remembered values, parameters, actions, modules, and aliases.

A name should start with a lowercase letter. It may contain lowercase letters, numbers, and underscores. It must not contain spaces and must not be an INTENTO keyword.

Valid names include `name`, `project_name`, `total_price`, and `item2`.

Example:

remember project_name as "INTENTO"

show project_name

Output:

INTENTO

Invalid names should produce syntax errors.

remember project name as "INTENTO"

Expected error:

Syntax error: names cannot contain spaces.

A name must exist before it is used.

show project_name

Expected error:

Semantic error: unknown name `project_name`.

Safety level: safe.

11.4 Output with `show`

The `show` instruction displays a value.

Syntax:

show value

The value may be text, a number, a boolean, `empty`, a list, a remembered name, an expression, or the result of an action that returns a value.

Example:

remember name as "Aurora"

show "Hello " plus name

Output:

Hello Aurora

The `show` instruction does not modify memory, files, folders, or external systems. It only sends information to output.

Invalid:

show

Expected error:

Syntax error: `show` needs a value.

Safety level: safe.

11.5 Memory with `remember`

The `remember` instruction stores a value in program memory.

Syntax:

remember name as value

Example:

remember language as "INTENTO"

show language

Output:

INTENTO

The `remember` instruction itself produces no visible output.

remember language as "INTENTO"

Output:

(no visible output)

A remembered value can be replaced by using the same name again.

remember status as "waiting"

show status

remember status as "running"

show status

Output:

waiting

running

Invalid:

remember name as

Expected error:

Syntax error: `remember` needs a value after `as`.

Safety level: safe.

11.6 Expressions and operators

An expression is code that produces a value.

INTENTO’s canonical operators are word operators: `plus`, `minus`, `times`, and `divided by`.

The operator `plus` adds numbers or joins text. The operators `minus`, `times`, and `divided by` require numbers.

Example:

show 10 plus 5

show 10 minus 5

show 10 times 5

show 10 divided by 5

Output:

15

5

50

2

Parentheses control evaluation order.

show 2 plus 3 times 4

show (2 plus 3) times 4

Output:

14

20

Division by zero is invalid at runtime.

show 10 divided by 0

Expected error:

Runtime error: division by zero.

Symbolic aliases such as `+`, `-`, `*`, and `/` may be accepted by Runtime versions as compact alternatives, but the canonical INTENTO form remains word-based. This rule is explained fully in the final appendix.

Safety level: safe.

11.7 Input with `ask`

The `ask` instruction receives input from the user and stores the answer under a name.

Syntax:

ask "Question?" and call the answer name

Example:

ask "What is your name?" and call the answer name

show "Hello " plus name

Example interaction:

What is your name?

Alessandro

Hello Alessandro

Input should normally be treated as text until explicitly converted.

The question must be written as text.

ask What is your name? and call the answer name

Expected error:

Syntax error: question text must be written between quotation marks.

Safety level: safe.

11.8 Conditions with `if` and `else`

The `if` structure runs a block only when a condition is true. The `else` structure runs an alternative block when the condition is false.

Syntax:

if condition:

```
    instruction
else:
    instruction
```

Example:

```
remember name as "Aurora"

if name is empty:
    show "Name is required"
else:
    show "Hello " plus name
```

Output:

```
Hello Aurora
```

With an empty value, the other path runs.

```
remember name as empty

if name is empty:
    show "Name is required"
else:
    show "Hello " plus name
```

Output:

```
Name is required
```

An `else` block must follow an `if` block.

```
else:
    show "No condition"
```

Expected error:

```
Syntax error: `else` must follow an `if` block.
```

Safety level: safe.

11.9 Comparisons

Comparisons are used inside conditions. Official comparison forms include `is`, `is not`, `is empty`, `is greater than`, `is less than`, `is at least`, and `is at most`.

Example:

```
remember age as 20


if age is at least 18:
    show "Access allowed"
else:
    show "Access denied"
```

Output:

```
Access allowed
```

Text comparison is also valid.

```
remember language as "INTENTO"


if language is "INTENTO":
    show "Correct"
else:
    show "Wrong"
```

Output:

```
Correct
```

Comparisons must respect types.

```
remember age as "20"


if age is at least 18:
```

show "Access allowed"

Expected error:

Type error: cannot compare text with number using `is at least`.

Safety level: safe.

11.10 Stopping execution with `stop`

The `stop` instruction ends program execution immediately.

Syntax:

stop

Example:

show "Start"

stop

show "End"

Output:

Start

The final instruction does not run.

The `stop` instruction does not take a value.

stop "now"

Expected error:

Syntax error: `stop` does not take a value.

Safety level: safe.

11.11 Repetition with `repeat`

The `repeat` instruction runs a block a fixed number of times.

Syntax:

repeat number times:

```
    instruction
```

Example:

```
repeat 3 times:
    show "Hello"
```

Output:

```
Hello
Hello
Hello
```

The repeat count must be numeric.

```
repeat "3" times:
    show "Hello"
```

Expected error:

```
Type error: repeat count must be a number.
```

A remembered number may be used.

```
remember attempts as 3


repeat attempts times:
    show "Trying..."
```

Output:

```
Trying...
Trying...
Trying...
```

Safety level: safe.

11.12 Lists and `for each`

The `for each` instruction processes every item in a list.

Syntax:

for each item in list:

    instruction

Example:

remember skills as ["Linux", "Python", "INTENTO"]

for each skill in skills:

    show skill

Output:

Linux

Python

INTENTO

The value after `in` must be a list.

remember skill as "Linux"

for each item in skill:

    show item

Expected error:

Type error: `for each` requires a list after `in`.

Safety level: safe.

11.13 Custom actions with `to`

The `to` instruction defines a custom action.

Defining an action does not run it. The action runs only when called.

Example:

to greet:

```
    show "Hello"


show "Ready"
```

Output:

```
Ready
```

Calling the action runs its block.

```
to greet:
    show "Hello"


greet
```

Output:

```
Hello
```

An action may receive parameters.

```
to greet with name:
    show "Hello " plus name


greet with "Aurora"
```

Output:

```
Hello Aurora
```

Calling an action with missing parameters is invalid.

```
to greet with name:
    show "Hello " plus name


greet
```

Expected error:

```
Action error: action `greet` needs 1 parameter.
```

Safety level: safe unless the action body contains confirmation-required or blocked instructions.

11.14 Returning values with `return`

The `return` instruction sends a value back from a custom action.

Syntax:

return value

Example:

to make_greeting with name:

    return "Hello " plus name

remember message as make_greeting with "Aurora"

show message

Output:

Hello Aurora

The `return` instruction can only be used inside a custom action.

return "Done"

Expected error:

Syntax error: `return` can only be used inside a custom action.

An action that does not return a value cannot be used as an expression.

to greet with name:

    show "Hello " plus name

remember message as greet with "Aurora"

Expected error:

Action error: action `greet` does not return a value.

Safety level: safe.

11.15 Conversion with `convert`

The `convert` instruction explicitly converts a value to another type.

Syntax:

convert value to type and call it name

Official target types are `text`, `number`, and `boolean`.

Example:

remember age_text as "42"

convert age_text to number and call it age

show age plus 1

Output:

43

Invalid conversion must fail clearly.

remember age_text as "forty-two"

convert age_text to number and call it age

Expected error:

Conversion error: cannot convert `forty-two` to number.

Boolean conversion should be strict.

remember answer as "true"

convert answer to boolean and call it active

show active

Output:

true

Invalid:

remember answer as "yes"

convert answer to boolean and call it active

Expected error:

Conversion error: cannot convert `yes` to boolean.

Safety level: safe.

11.16 Error handling with `try` and `on error`

The `try` and `on error` structure handles recoverable errors.

Syntax:

```
try:
    instruction
on error:
    instruction
```

Example:

```
remember age_text as "abc"

try:
    convert age_text to number and call it age
    show age plus 1
on error:
    show "Age must be a number"
```

Output:

Age must be a number

With valid input, the error block does not run.

```
remember age_text as "42"


try:
    convert age_text to number and call it age
    show age plus 1
on error:
    show "Age must be a number"
```

Output:

```
43
```

Safety violations caused by blocked operations should not normally be recoverable through `try`.

```
try:
    run shell "some command"
on error:
    show "Ignored"
```

Expected error:

```
Safety error: raw shell execution is not allowed in INTENTO.
```

Safety level: safe for ordinary recoverable errors.

11.17 Confirmation with `with confirmation`

The `with confirmation` block marks sensitive operations that require user approval.

Syntax:

```
with confirmation:
    instruction
```

Example:

with confirmation:

    create folder "Projects/Test"

show "Done"

Expected confirmation prompt:

Confirm operation: create folder Projects/Test?

If confirmed, output:

Done

If refused, possible output:

Operation cancelled by user.

Confirmation is not a replacement for safety rules. Blocked operations remain blocked.

Safety level: confirmation required.

11.18 File and folder instructions

INTENTO defines a small set of controlled file and folder instructions.

The instruction `create folder` creates a folder.

with confirmation:

    create folder "Projects/INTENTO"

show "Folder ready"

Expected confirmation prompt:

Confirm operation: create folder Projects/INTENTO?

If confirmed, output:

Folder ready

Expected filesystem result:

Projects/INTENTO/

The instruction `create file` creates a file with content.

with confirmation:

```
create file "notes.txt" with "Hello"
```

show "File created"

Expected confirmation prompt:

Confirm operation: create file notes.txt?

If confirmed, output:

File created

Expected file content:

Hello

The instruction `read file` reads file content into memory.

read file "notes.txt" and call it notes

show notes

Expected output if `notes.txt` contains `Hello`:

Hello

The instruction `append to file` adds content to the end of an existing file.

with confirmation:

```
append to file "log.txt" with "Program started"
```

show "Log updated"

Expected confirmation prompt:

Confirm operation: append to file log.txt?

If confirmed, output:

Log updated

Deletion and unrestricted overwrite are not available in Draft 0.1.

delete file "notes.txt"

Expected error:

Safety error: deletion is not available in INTENTO Draft 0.1.

Safety level: reading is safe only inside the allowed workspace; creation and append require confirmation; deletion is blocked.

11.19 Modules

The `use module` instruction imports reusable INTENTO code from another `.intento` file.

Syntax:

use module "path"

Example:

use module "formatting.intento"

show_title with "INTENTO"

Expected output if `formatting.intento` defines `show_title`:

== INTENTO ==

Modules may also be imported with aliases.

use module "formatting.intento" as formatting

formatting.show_title with "INTENTO"

Expected output:

== INTENTO ==

If the module is missing, execution should fail clearly.

use module "missing.intento"

Expected error:

Module error: module not found: missing.intento.

Safety level: safe if the module path is inside the allowed workspace.

11.20 Libraries and registered actions

The `use library` instruction loads a registered library.

Syntax:

use library "name"

Optional version syntax:

use library "name" version "1.0"

Example:

use library "text"

show "Library loaded"

Output:

Library loaded

A registered action is called with `use action`.

Syntax when the action returns a value:

use action "namespace.action" with value and call the result name

Example:

use library "text"

remember title as "Human Readable Code"

use action "text.slugify" with title and call the result slug

show slug

Output:

human-readable-code

Syntax when the action performs an operation without returning a value:

use action "namespace.action" with value

Example:

use library "system"

use action "system.notify" with "Task complete"

show "Done"

Possible output:

Done

Registered actions must be known to the Runtime. They must declare accepted parameter types, return type, safety level, and description.

Invalid:

use action "text.unknown" with "Hello" and call the result value

Expected error:

Semantic error: unknown registered action `text.unknown`.

Safety level: defined by action metadata.

11.21 Action metadata with `describe action`

The `describe action` instruction displays metadata for a registered action.

Syntax:

describe action "namespace.action"

Example:

use library "text"

describe action "text.slugify"

Possible output:

Action: text.slugify

Accepts: text

Returns: text

Safety: safe

Description: Converts text into a lowercase URL-safe slug.

If the action does not exist, the Runtime should report an error.

describe action "text.unknown"

Expected error:

Semantic error: unknown registered action `text.unknown`.

Safety level: safe.

11.22 Namespaces

A namespace groups actions and prevents name conflicts.

In the action name `text.slugify`, the namespace is `text` and the action is `slugify`.

Namespaces make larger programs clearer.

Example:

use library "text"

use library "list"

remember skills as ["Linux", "Python", "INTENTO"]

use action "list.count" with skills and call the result total

use action "text.slugify" with "Human Readable Code" and call the result slug

show total

show slug

Output:

3

human-readable-code

If an action name is ambiguous, the Runtime should require a namespaced name.

use action "format" with "value" and call the result result

Expected error:

Semantic error: ambiguous action `format`. Use a namespaced action name.

Safety level: safe.

11.23 Safety levels

Every INTENTO instruction belongs to a safety level.

Safe instructions do not modify the user's files, folders, operating system, network, or external services. Examples include `show`, `remember`, `ask`, `if`, `else`, `repeat`, `for each`, `to`, `return`, `convert`, and ordinary expressions.

Confirmation-required instructions can modify the user's environment. Examples include `create folder`, `create file`, `append to file`, and registered actions that write files, access the network, or modify external systems.

Blocked operations are not allowed in Draft 0.1. Examples include raw shell execution, raw Python code inside INTENTO source, unrestricted deletion, unrestricted overwrite, paths outside the allowed workspace, and unregistered backend actions.

Example of blocked raw shell execution:

run shell "ls"

Expected error:

Safety error: raw shell execution is not allowed in INTENTO.

Example of blocked raw Python code:

use python code "print('hello')"

Expected error:

Safety error: raw Python code is not allowed in INTENTO programs.

Confirmation does not make blocked operations valid.

11.24 Error categories

Official error categories include syntax error, semantic error, type error, conversion error, file error, module error, library error, action error, runtime error, safety error, and project error.

A good error message should identify the category and the problem. When possible, it should also identify the line.

Example:

show unknown_value

Expected error:

Semantic error: unknown name `unknown_value`.

Better diagnostic:

Semantic error on line 1: unknown name `unknown_value`.

The Runtime should avoid vague messages. INTENTO is intended to be easy to learn, and its errors should help the programmer understand the rule that was broken.

11.25 Scope rules

Top-level remembered values belong to main program memory. Custom actions have local scope. Parameters and values remembered inside an action exist only inside that action unless returned.

Example:

```
to make_message with name:
    remember message as "Hello " plus name
    return message


remember result as make_message with "Aurora"
show result
```

Output:

Hello Aurora

The local value `message` is not available outside the action.

to make_message with name:

    remember message as "Hello " plus name

    return message

show message

Expected error:

Semantic error: unknown name `message`.

Actions should receive values through parameters and return values explicitly. This keeps dependencies visible.

Safety level: safe.

11.26 Execution rules

The Runtime should parse the source file before execution. If syntax is invalid, execution should not begin.

The Runtime should check names, actions, types, and safety rules as early as reasonably possible. If a semantic or type error is known before execution, the program should not partially run.

During execution, instructions run from top to bottom unless control flow changes through `if`, `else`, `repeat`, `for each`, action calls, `return`, `stop`, or errors.

Unhandled errors stop execution.

Example:

show "Start"

show 10 divided by 0

show "End"

Expected output:

Start

Runtime error: division by zero.

The final instruction does not run.

The `stop` instruction is not an error. It is intentional control flow.

show "Start"

stop

show "End"

Output:

Start

11.27 Project metadata

A larger INTENTO project may include a metadata file named `intento.project`.

This file is not executable INTENTO code. It describes project requirements such as Runtime version, libraries, permissions, and workspace rules.

Possible content:

runtime: INTENTO 1.0

libraries: text, list, file

permissions: read project files, write project files with confirmation

If a required library is missing, the Runtime should fail before execution.

Expected error:

Project error: required library `text` is not available.

Project metadata makes INTENTO programs easier to distribute and safer to execute.

Safety level: safe.

11.28 Complete reference example

This example uses several official language features together.

use library "text"

show "Project initializer"

ask "Project title?" and call the answer title

```
if title is empty:
    show "Project title is required"
    stop

use action "text.slugify" with title and call the result slug

remember folder as "Projects/" plus slug
remember readme as folder plus "/README.txt"
remember readme_text as "# " plus title

with confirmation:
    create folder folder
    create file readme with readme_text

show "Project created: " plus folder
```

Example interaction:

```
Project initializer
Project title?
Human Terminal
Confirm operation: create folder Projects/human-terminal?
Confirm operation: create file Projects/human-terminal/README.txt?
Project created: Projects/human-terminal
```

Expected filesystem result:

```
Projects/human-terminal/
Projects/human-terminal/README.txt
```

Expected README content:

Human Terminal

This example uses `use library`, `show`, `ask`, `if`, `stop`, `use action`, `remember`, `plus`, `with confirmation`, `create folder`, and `create file`.

It represents the intended style of INTENTO: readable source, explicit validation, controlled backend power, and safe execution.

11.29 Reserved features

The following capabilities are not official in Draft 0.1, but may be introduced in future versions: `while`, `repeat until`, `and`, `or`, `not`, dictionary values, record values, classes, async tasks, package management, database actions, HTTP actions, testing syntax, debug commands, overwrite operations, and deletion operations.

Future features must follow the same requirements as existing features. They must have readable syntax, precise parsing rules, clear runtime behavior, expected output or expected errors, and a defined safety level.

INTENTO should grow carefully. A command should not become official only because it is useful. It becomes official when it is useful, readable, precise, and safe.

11.30 Final reference rule

INTENTO is not free natural language. It is not raw Python. It is not shell scripting with softer words.

INTENTO is a human-readable programming language with controlled syntax and real execution.

Every official command must answer five questions.

What is the syntax?
What does it mean?
What value does it produce, if any?
What output or error should be expected?
What safety level does it have?

If a command cannot answer these questions, it does not belong in the official language yet.

Chapter 12 — Standard Library Reference

The INTENTO language should remain small at its core.

A small core is easier to learn, easier to parse, and easier to implement safely. But a real language also needs practical tools. Programs often need to process text, count items, work with dates, read structured data, write logs, or call controlled system capabilities.

This is the role of the INTENTO Standard Library.

The Standard Library is the official collection of registered actions provided by the Runtime. These actions are not raw backend code. They are named, typed, documented capabilities that INTENTO programs can call through controlled syntax.

The core language defines how programs are written.
The Standard Library defines what common operations are available.

The important rule is simple: library power must remain readable and safe.

12.1 What a standard library action is

A standard library action is a registered action that belongs to an official namespace.

For example:

text.slugify

list.count

date.format

json.get

log.info

The part before the dot is the namespace. The part after the dot is the action name.

A standard action must define what it accepts, what it returns, and what safety level it has. If an action writes files, accesses the network, or modifies the user’s environment, the Runtime must know that before execution.

A returning action uses this form:

use action "namespace.action" with value and call the result name

Example:

use library "text"

remember title as "Human Readable Code"

use action "text.slugify" with title and call the result slug

show slug

Output:

human-readable-code

The action receives the text value stored in `title`, produces a new text value, and stores that result as `slug`.

An action that does not return a value uses this form:

use action "namespace.action" with value

Example:

use library "log"

use action "log.info" with "Program started"

show "Running"

Output:

Running

Expected log entry:

INFO Program started

The visible output comes from `show`. The logging action writes to the Runtime log.

If an action has no parameters, the `with` part may be omitted:

use action "namespace.action" and call the result name

Example:

use library "date"

use action "date.today" and call the result today

show today

Possible output:

2026-06-13

The exact date depends on the system date used by the Runtime.

12.2 Library loading

A library must be loaded before its actions are used.

Syntax:

use library "name"

Example:

use library "text"

show "Text library loaded"

Output:

Text library loaded

Loading a library does not normally display output. The visible output in this example comes from `show`.

If the library does not exist, the Runtime should report a clear error.

use library "unknown"

show "Ready"

Expected error:

Library error: unknown library `unknown`.

A library may also be loaded with a version.

use library "text" version "1.0"

show "Text library 1.0 loaded"

Output:

Text library 1.0 loaded

If the requested version is not available, execution should fail before the program uses the library.

use library "text" version "99.0"

Expected error:

Library error: library `text` version `99.0` is not available.

Versioning is not necessary for every small program, but it is important for stable programs that must behave the same over time.

12.3 The `text` library

The `text` library provides common text operations.

Official basic actions may include `text.trim`, `text.uppercase`, `text.lowercase`, `text.slugify`, `text.count_words`, and `text.split`.

The action `text.trim` removes leading and trailing spaces.

use library "text"

remember title as " INTENTO Manual "

use action "text.trim" with title and call the result clean_title

show clean_title

Output:

INTENTO Manual

The action `text.slugify` converts text into a lowercase, path-friendly form.

use library "text"

remember title as "Human Readable Code"

use action "text.slugify" with title and call the result slug

show slug

Output:

human-readable-code

The action `text.count_words` returns the number of words in a text value.

use library "text"

remember sentence as "INTENTO is readable code"

use action "text.count_words" with sentence and call the result count

show count

Output:

4

The `text` actions require text input unless the action explicitly documents otherwise.

Invalid:

use library "text"

remember value as 42

use action "text.count_words" with value and call the result count

show count

Expected error:

Type error: action `text.count_words` expects text.

Safety level: safe. Text actions transform values in memory and do not modify the external system.

12.4 The `number` library

The core language already supports basic arithmetic with `plus`, `minus`, `times`, and `divided by`. The `number` library provides practical numeric helpers that are useful but do not need to be core syntax.

Possible official actions include `number.round`, `number.absolute`, `number.minimum`, and `number.maximum`.

Example:

use library "number"

remember value as -12

use action "number.absolute" with value and call the result positive

show positive

Output:

12

The action `number.round` may round a decimal number to the nearest whole number.

use library "number"

remember value as 12.7

use action "number.round" with value and call the result rounded

show rounded

Output:

13

The `number.minimum` action can return the smaller of two numbers.

```
use library "number"

remember first as 10
remember second as 5

use action "number.minimum" with first and second and call the result smaller

show smaller
```

Output:

```
5
```

Number actions should reject text values that only look numeric.

```
use library "number"

remember value as "12"

use action "number.absolute" with value and call the result positive

show positive
```

Expected error:

```
Type error: action `number.absolute` expects number.
If text must become a number, the program should use explicit conversion first.
Safety level: safe.
```

12.5 The `list` library

The core language supports list values and `for each`. The `list` library provides common list operations.

Possible official actions include `list.count`, `list.join`, `list.contains`, `list.first`, and `list.last`.

The action `list.count` returns the number of items in a list.

```
use library "list"

remember skills as ["Linux", "Python", "INTENTO"]

use action "list.count" with skills and call the result total

show total
```

Output:

```
3
```

The action `list.join` turns a list into text by placing a separator between items.

```
use library "list"

remember skills as ["Linux", "Python", "INTENTO"]

use action "list.join" with skills and ", " and call the result skill_text

show skill_text
```

Output:

```
Linux, Python, INTENTO
```

The action `list.contains` checks whether a list contains a value and returns a boolean.

```
use library "list"

remember skills as ["Linux", "Python", "INTENTO"]

use action "list.contains" with skills and "Python" and call the result has_python
```

```
if has_python is true:
    show "Python found"
else:
    show "Python missing"
```

Output:

```
Python found
```

List actions should require actual lists.

Invalid:

```
use library "list"

remember skills as "Linux, Python, INTENTO"

use action "list.count" with skills and call the result total

show total
```

Expected error:

```
Type error: action `list.count` expects list.
```

A text value containing commas is not the same as a list. If the programmer wants to split text into a list, that should be done with a documented text action such as `text.split`.

Safety level: safe.

12.6 The `file` library

INTENTO already has core file instructions such as `read file`, `create file`, `create folder`, and `append to file`. The `file` library provides helper actions around files and paths.

The `file` library should not bypass the safety model. It must obey workspace restrictions and confirmation rules.

A safe helper is `file.exists`, which checks whether a file exists inside the allowed workspace.

```
use library "file"

use action "file.exists" with "notes.txt" and call the result exists

if exists is true:
    show "File exists"
else:
    show "File missing"
```

Possible output:

```
File exists
```

The output depends on whether `notes.txt` exists in the allowed workspace.

A path outside the allowed workspace should be rejected.

```
use library "file"

use action "file.exists" with "/system/private.txt" and call the result exists

show exists
```

Expected error:

```
Safety error: path is outside the allowed workspace.
```

The `file.name` action may extract the file name from a path.

```
use library "file"

use action "file.name" with "Projects/INTENTO/README.txt" and call the result file_name

show file_name
```

Output:

README.txt

The `file.extension` action may extract the extension.

```
use library "file"

use action "file.extension" with "notes.txt" and call the result extension

show extension
```

Output:

```
txt
```

Actions that write files, move files, or modify folders must require confirmation. File helper actions that only inspect paths or check existence may be safe, but only inside the allowed workspace.

Safety level: safe for read-only helpers inside the allowed workspace; confirmation required for modification; blocked outside the allowed workspace.

12.7 The `date` library

Dates are useful, but date rules can become complicated. They depend on formats, locales, time zones, and calendar conventions. For this reason, date operations belong in the Standard Library, not in the core syntax.

The `date.today` action returns the current date according to the Runtime environment.

```
use library "date"

use action "date.today" and call the result today

show "Today: " plus today
```

Possible output:

```
Today: 2026-06-13
```

The exact output depends on the Runtime date.

The `date.format` action can format a date string.

```
use library "date"
```

```
remember raw_date as "2026-06-13"

use action "date.format" with raw_date and "long" and call the result readable_date

show readable_date
```

Possible output:

```
June 13, 2026
```

The format names supported by the Runtime must be documented. Common formats may include `"iso"`, `"short"`, and `"long"`.

Invalid dates should not be guessed.

```
use library "date"

remember raw_date as "not-a-date"

use action "date.format" with raw_date and "long" and call the result readable_date

show readable_date
```

Expected error:

```
Date error: cannot parse date `not-a-date`.
```

A strict date library is important because ambiguous dates can cause serious mistakes. The text `"01/02/2026"` may mean different things in different places. INTENTO should prefer explicit formats.

Safety level: safe.

12.8 The `json` library

JSON is a common format for structured data. INTENTO should not make JSON part of the core syntax. Instead, JSON should be handled by a standard library.

The `json.get` action reads a value from JSON text using a key.

```
use library "json"

remember data as "{\"name\":\"Aurora\",\"language\":\"INTENTO\"}"

use action "json.get" with data and "name" and call the result name
use action "json.get" with data and "language" and call the result language

show "Name: " plus name
show "Language: " plus language
```

Output:

```
Name: Aurora
Language: INTENTO
```

The action receives JSON text and a key. It returns the value associated with that key.

If the JSON text is invalid, the Runtime should report an error.

```
use library "json"

remember data as "{bad json}"

use action "json.get" with data and "name" and call the result name

show name
```

Expected error:

```
JSON error: invalid JSON text.
```

If the key does not exist, the action should fail clearly unless a different action is explicitly designed to return `empty`.

```
use library "json"
```

```
remember data as "{\"name\":\"Aurora\"}"

use action "json.get" with data and "language" and call the result language

show language
```

Expected error:

```
JSON error: key `language` not found.
```

A separate action such as `json.has_key` may be used before reading a key.

```
use library "json"

remember data as "{\"name\":\"Aurora\"}"

use action "json.has_key" with data and "name" and call the result has_name

if has_name is true:
    show "Name exists"
else:
    show "Name missing"
```

Output:

```
Name exists
```

Safety level: safe. JSON actions process values in memory and do not access files unless combined with file instructions.

12.9 The `csv` library

CSV is another common data format. It is often used for tables, exports, reports, and spreadsheets.

CSV may look simple, but correct CSV parsing is not the same as splitting text by commas. Quoted fields, line endings, and headers must be handled according to documented rules. For this reason, CSV support belongs in the Standard Library.

A basic action is `csv.count_rows`.

Example file content for `people.csv`:

```
name,role
Ada,mathematician
Grace,computer scientist
```

Program:

```
use library "csv"

read file "people.csv" and call it csv_text

use action "csv.count_rows" with csv_text and call the result rows

show "Rows: " plus rows
```

Output:

```
Rows: 2
```

In this definition, the header row is not counted as a data row. That rule must be documented by the action.

Another useful action is `csv.get_column`, which returns a list of values from a named column.

```
use library "csv"

read file "people.csv" and call it csv_text

use action "csv.get_column" with csv_text and "name" and call the result names

show names
```

Output:

```
["Ada", "Grace"]
```

If the CSV text is invalid, the Runtime should report a CSV error.

```
use library "csv"

remember csv_text as "bad,csv,\"broken"

use action "csv.count_rows" with csv_text and call the result rows

show rows
```

Expected error:

```
CSV error: invalid CSV text.
```

If the requested column does not exist, the Runtime should report a clear error.

```
use library "csv"

read file "people.csv" and call it csv_text

use action "csv.get_column" with csv_text and "age" and call the result ages

show ages
```

Expected error:

```
CSV error: column `age` not found.
```

Safety level: safe for parsing text in memory. Reading the CSV file itself follows the safety level of `read file`.

12.10 The `log` library

Logs record what happened during execution.

Visible output is for the user. Logs are for diagnosis, auditing, and debugging. A program may show only a simple message to the user while recording more detailed information in the Runtime log.

The `log.info` action records an informational message.

```
use library "log"

use action "log.info" with "Program started"

show "Started"
```

Output:

```
Started
```

Expected log entry:

```
INFO Program started
```

The `log.warning` action records a warning.

```
use library "log"

use action "log.warning" with "Using default configuration"

show "Configuration loaded"
```

Output:

```
Configuration loaded
```

Expected log entry:

```
WARNING Using default configuration
```

The `log.error` action records an error message without necessarily stopping execution.

```
use library "log"

use action "log.error" with "Input file missing"

show "Could not continue"
```

Output:

Could not continue

Expected log entry:

ERROR Input file missing

Logging actions should not expose sensitive data automatically. They only log the values explicitly given by the program.

Safety level: safe when writing to the Runtime log. If a Runtime allows logging to user-selected files, file safety rules apply.

12.11 The `system` library

The `system` library exposes controlled local system actions.

This namespace must be treated carefully. It should not become a back door for arbitrary system control. Every `system` action must be registered, documented, typed, and safety-checked.

A simple example is `system.notify`, which asks the operating system to show a notification.

```
use library "system"



with confirmation:

    use action "system.notify" with "INTENTO task complete"



show "Notification requested"
```

Expected confirmation prompt:

Confirm operation: system.notify?

If confirmed, output:

Notification requested

Expected side effect:

A system notification is shown with the text "INTENTO task complete".

Depending on Runtime configuration, notifications may be considered safe or confirmation-required. A strict Runtime should require confirmation because the action interacts with the user’s desktop environment.

Raw system execution must remain blocked.

use library "system"

use action "system.run_raw" with "some command"

Expected error:

Safety error: raw system execution is not allowed.

The `system` library should expose only controlled actions, not unrestricted shell access.

Safety level: depends on action metadata; raw execution is blocked.

12.12 The `network` library

The `network` library provides controlled access to external resources.

Network access is sensitive because it communicates outside the local machine. It may reveal information, depend on remote services, or produce results that change over time. For this reason, network actions should require explicit Runtime permission and usually confirmation.

A basic action is `network.fetch_text`.

use library "network"

with confirmation:

    use action "network.fetch_text" with "https://example.com/status" and call the result status

show status

Expected confirmation prompt:

Confirm operation: network.fetch_text https://example.com/status?

Possible output if the remote service returns `OK`:

OK

If network access is disabled, the Runtime should block the action.

use library "network"

with confirmation:

```
    use action "network.fetch_text" with "https://example.com/status" and call the result status


show status
```

Expected error in a restricted Runtime:

Safety error: network access is not allowed in this Runtime configuration.

Network actions should document what they send, what they receive, and whether they store anything. A readable language must be especially clear when it communicates outside the machine.

Safety level: confirmation required and permission-dependent.

12.13 Standard file helpers versus core file commands

INTENTO has both core file commands and file library helpers.

Core file commands express direct file intentions:

read file

create file

create folder

append to file

The `file` library provides helper actions that support file-related programs:

file.exists

file.name

file.extension

A good program may use both.

```
use library "file"


use action "file.exists" with "notes.txt" and call the result exists
```

if exists is true:

    read file "notes.txt" and call it notes

    show notes

else:

    show "notes.txt does not exist"

Possible output if the file exists and contains `Hello`:

Hello

Possible output if the file does not exist:

notes.txt does not exist

This structure is clearer than trying to read the file first and letting the program fail. It asks a question, then follows the correct path.

Safety level: the helper action is safe inside the allowed workspace; `read file` follows file-read safety rules.

12.14 Standard library error rules

Standard library actions should use the same error categories as the rest of INTENTO.

If the action name is unknown, the Runtime should report a semantic error.

use library "text"

use action "text.unknown" with "Hello" and call the result value

show value

Expected error:

Semantic error: unknown registered action `text.unknown`.

If the action receives the wrong type, the Runtime should report a type error.

use library "list"

remember value as "not a list"

use action "list.count" with value and call the result total

show total

Expected error:

Type error: action `list.count` expects list.

If a format-specific action fails, it should use a format-specific error when useful.

use library "json"

remember data as "{bad json}"

use action "json.get" with data and "name" and call the result name

show name

Expected error:

JSON error: invalid JSON text.

If an action violates a safety rule, the Runtime should report a safety error.

use library "network"

use action "network.fetch_text" with "https://example.com" and call the result page

show page

Expected error in a restricted Runtime:

Safety error: network access is not allowed in this Runtime configuration.

Errors should be precise. The programmer should know whether the problem is an unknown action, a wrong type, invalid data, or a blocked operation.

12.15 A complete standard library example

This example uses several standard library namespaces together.

It reads a note file, trims the content, counts words, creates a slug from the title, writes a small report, and logs the operation.

Expected content of `notes.txt`:

INTENTO is human readable code

Program:

```
use library "text"

use library "file"

use library "log"


show "Note report"


use action "file.exists" with "notes.txt" and call the result exists


if exists is false:
    show "notes.txt is missing"
    stop


read file "notes.txt" and call it notes


use action "text.trim" with notes and call the result clean_notes

use action "text.count_words" with clean_notes and call the result words

use action "text.slugify" with "Note Report" and call the result report_name


remember report_path as report_name plus ".txt"
```

```
remember report_text as "Words: " plus words


with confirmation:
    create file report_path with report_text


use action "log.info" with "Note report created"


show "Report created: " plus report_path
```

Example interaction if the file exists and the user confirms:

```
Note report
Confirm operation: create file note-report.txt?
Report created: note-report.txt
```

Expected created file:

```
note-report.txt
```

Expected file content:

```
Words: 5
```

Expected log entry:

```
INFO Note report created
```

If `notes.txt` is missing, the output is:

```
Note report
notes.txt is missing
```

The program shows the intended style of the Standard Library. It loads only the libraries it needs. It checks whether the file exists before reading it. It uses text actions for text processing. It uses confirmation before creating a file. It records a log entry after the operation.

The code remains readable, while the Runtime handles the technical work through registered actions.

12.16 Summary

The INTENTO Standard Library extends the language without making the core syntax heavy.

The `text` library handles text transformation and analysis. The `number` library provides numeric helpers. The `list` library provides list operations. The `file` library provides safe file-related helpers. The `date`, `json`, and `csv` libraries handle common structured data. The `log` library records execution events. The `system` and `network` libraries expose controlled external capabilities under strict safety rules.

Every standard action must have a clear name, accepted input types, a return type or side effect, documented behavior, and a safety level.

The Standard Library should not turn INTENTO into hidden Python or hidden shell execution. It should expose practical power through readable, registered, controlled actions.

The purpose of the Standard Library is not to make INTENTO larger for its own sake.

Its purpose is to let real programs stay simple.

Chapter 13 — Runtime, Project Format, and Implementation Rules

A language is complete only when it can be executed consistently.

The previous chapters defined the INTENTO syntax, the Standard Library, modules, registered actions, safety rules, and error behavior. This final technical chapter explains how those rules become a practical system.

The INTENTO Runtime is the program that reads `.intento` files, checks them, applies safety rules, executes instructions, calls registered actions, produces output, and records what happened.

The goal of the Runtime is not only to “run code.” Its goal is to preserve the identity of INTENTO during execution: readable intention, precise parsing, controlled power, and safe behavior.

A correct Runtime should not guess. It should parse, validate, execute, report, and log.

13.1 What the Runtime is

The Runtime is the execution engine of INTENTO.

It receives an INTENTO source file, reads its instructions, checks the structure, verifies names and types, applies the safety policy, and executes the program.

A simple command may look like this:

intento run hello.intento

If `hello.intento` contains:

show "Hello from INTENTO"

Expected terminal output:

Hello from INTENTO

This looks simple from the user’s point of view, but several steps happen internally. The Runtime reads the file, tokenizes the source, parses the instructions, checks that `show` is valid, checks that `"Hello from INTENTO"` is a valid text value, and then sends the value to output.

The user sees the result.
The Runtime enforces the rules.

This separation is important. INTENTO source code should remain simple, while the Runtime carries the technical responsibility underneath.

13.2 Execution pipeline

A correct INTENTO Runtime should follow a clear execution pipeline.

The source file is read first. Then the Runtime parses the program. After parsing, it performs semantic checks, type checks, safety checks, and dependency checks. Only after those steps should execution begin.

The pipeline can be described like this:

source file

parser

semantic checker

type checker

safety engine

runtime executor

registered actions

output and logs

For example:

show "Start"

show unknown_value

show "End"

Expected error:

Semantic error: unknown name `unknown_value`.

A strict Runtime should detect this before running the program. It should not print `Start` and then fail halfway if the missing name can already be known before execution.

However, some errors happen only during execution.

show "Start"

remember divisor as 0

show 10 divided by divisor

show "End"

Expected output:

Start

Runtime error: division by zero.

The final line does not run. The program was structurally valid, but execution reached an impossible operation.

The Runtime should distinguish between errors that can be detected before execution and errors that occur during execution. This makes diagnostics clearer and behavior safer.

13.3 Source files and entry point

The main source file of an INTENTO program should normally be named with the `.intento` extension.

Examples:

main.intento

hello.intento

project_setup.intento

A project may contain many `.intento` files, but one file is the entry point: the file passed to the Runtime.

Example:

intento run main.intento

Expected behavior:

The Runtime executes main.intento as the main program.

If the file does not exist, the Runtime should fail clearly.

intento run missing.intento

Expected error:

Runtime error: source file not found: missing.intento.

If the file extension is wrong, the Runtime may reject it in strict mode.

intento run main.txt

Expected error in strict mode:

Runtime error: INTENTO source files must use the .intento extension.

A permissive development Runtime may allow other text files for testing, but the official project format should prefer `.intento` files. This keeps projects clear and recognizable.

13.4 Project structure

A larger INTENTO project should have a predictable structure.

A simple project may look like this:

my_project/

main.intento

modules/

data/

output/

intento.project

README.txt

The file `main.intento` is the entry point. The `modules` folder contains reusable INTENTO modules. The `data` folder contains input files. The `output` folder may contain generated files. The `intento.project` file declares project metadata. The `README.txt` explains the project for humans.

The Runtime should not require every project to use this full structure. A single file should still be valid.

Example:

hello.intento

If `hello.intento` contains:

show "Hello"

Running it should produce:

Hello

Small programs should stay small. Project structure becomes useful when programs grow.

The important rule is that larger programs should make their dependencies and permissions visible. INTENTO should not hide project behavior in scattered files without explanation.

13.5 The `intento.project` file

The optional `intento.project` file describes the project.

It is not executable INTENTO code. It is metadata for the Runtime and for the user.

A simple `intento.project` file may contain:

runtime: INTENTO 1.0

entry: main.intento

libraries: text, list, file

permissions: read project files, write project files with confirmation

This tells the Runtime which version is expected, which file is the entry point, which libraries are needed, and what permissions the project requests.

If a required library is missing, the Runtime should fail before execution.

Expected error:

Project error: required library `text` is not available.

If the project requests a Runtime version that is not supported, the Runtime should also fail clearly.

Example metadata:

runtime: INTENTO 9.0

entry: main.intento

Expected error:

Project error: required Runtime version INTENTO 9.0 is not supported.

Project metadata is part of making INTENTO practical. A program should not only say what it does inside the source file. It should also declare what environment it needs.

13.6 Workspace rules

The workspace is the area of the filesystem where an INTENTO program is allowed to operate.

By default, the workspace should be the project folder or the folder from which the program is executed. This prevents programs from reading or writing arbitrary parts of the user’s system.

Example:

read file "data/notes.txt" and call it notes

show notes

Expected output if the file exists inside the workspace and contains `Hello`:

Hello

This is allowed because the path is relative to the workspace.

A path outside the workspace should be rejected.

read file "/system/private.txt" and call it data

show data

Expected error:

Safety error: path is outside the allowed workspace.

The same rule applies to writing.

with confirmation:

    create file "/system/config.txt" with "change"

Expected error:

Safety error: path is outside the allowed workspace.

Confirmation does not override workspace rules. A confirmed unsafe path is still unsafe.

A Runtime may allow the user to configure additional allowed folders, but those folders must be explicit. INTENTO should never assume that the whole computer is available.

13.7 Permissions

Permissions define what a program is allowed to do.

A simple program that only uses `show`, `remember`, conditions, loops, and text processing may require no special permissions.

A program that reads project files may require read permission. A program that creates files may require write permission and confirmation. A program that accesses the network may require network permission. A program that sends notifications may require system interaction permission.

A project metadata file may declare permissions like this:

permissions: read project files, write project files with confirmation

If a program tries to perform an action without the required permission, the Runtime should report a safety error.

Example:

with confirmation:

    create file "output/report.txt" with "Report"

Expected error if write permission is not granted:

Safety error: project does not have permission to write files.

If permission is granted and the user confirms, the operation may proceed.

Expected confirmation prompt:

Confirm operation: create file output/report.txt?

If confirmed, expected result:

output/report.txt

Permissions make INTENTO safer because they allow the Runtime to reject unexpected capabilities before they are used.

13.8 Confirmation behavior

The `with confirmation` block gives the user a chance to approve sensitive operations.

A Runtime may ask for confirmation for each operation or summarize a group of operations. Both approaches are acceptable if the user clearly understands what will happen.

Example:

with confirmation:

```
    create folder "Projects/Test"

    create file "Projects/Test/README.txt" with "Test project"


show "Project prepared"
```

Possible confirmation prompts:

Confirm operation: create folder Projects/Test?

Confirm operation: create file Projects/Test/README.txt?

If confirmed, output:

Project prepared

Expected filesystem result:

Projects/Test/

Projects/Test/README.txt

A Runtime may instead summarize the block:

Confirm 2 operations?

- create folder Projects/Test

- create file Projects/Test/README.txt

If the user refuses, the Runtime should not perform the operation.

Possible output:

Operation cancelled by user.

If an operation is cancelled, the Runtime should define whether the program stops or continues. The recommended rule is simple: cancellation of a required operation stops execution unless the operation is inside a `try` block that can handle ordinary cancellation.

This prevents programs from continuing after a file or folder they needed was not created.

13.9 Runtime modes

A Runtime may support different execution modes.

A normal mode runs the program with standard safety checks and ordinary output.

A check mode validates the program without executing it.

Example:

intento check main.intento

Possible output:

Program check passed.

If the program has an error:

Semantic error on line 4: unknown name `project`.

A dry-run mode shows what would happen without modifying files or external systems.

Example:

intento dry-run main.intento

Possible output:

Would create folder Projects/Test

Would create file Projects/Test/README.txt

Dry run complete.

A debug mode may show additional information such as parsed instructions, loaded modules, libraries, permissions, and action calls.

Example:

intento run main.intento --debug

Possible output:

Loaded library: text

Loaded module: modules/reporting.intento

Executing: show

Executing: use action text.slugify

Program finished.

Runtime modes are useful because they let programmers inspect behavior before real execution. This is especially important for a language that can affect files and external systems.

13.10 CLI behavior

The command-line interface should be small and predictable.

Recommended commands include:

intento run file.intento

intento check file.intento

intento dry-run file.intento

intento describe action namespace.action

intento list libraries

intento list actions

Running a program:

intento run hello.intento

Expected output if `hello.intento` shows `Hello`:

Hello

Checking a program:

intento check hello.intento

Possible output:

Program check passed.

Listing libraries:

intento list libraries

Possible output:

text

number

list

file

date

json

csv

log

system

network

Describing an action:

intento describe action text.slugify

Possible output:

Action: text.slugify

Accepts: text

Returns: text

Safety: safe

Description: Converts text into a lowercase URL-safe slug.

The CLI should not be overloaded. Its purpose is to run, check, inspect, and diagnose INTENTO programs.

13.11 Logs

The Runtime should keep an execution log.

Logs are useful for debugging, auditing, and understanding what happened. They should include important events such as program start, loaded libraries, loaded modules, confirmation decisions, registered action calls, file operations, errors, and program end.

A simple program:

use library "text"

show "Start"

remember title as "Human Readable Code"

use action "text.slugify" with title and call the result slug

show slug

Output:

Start

human-readable-code

Possible Runtime log:

Program started: main.intento

Loaded library: text

Called action: text.slugify

Program finished successfully.

If an error occurs:

show unknown_value

Expected error:

Semantic error: unknown name `unknown_value`.

Possible Runtime log:

Program started: main.intento

Semantic error on line 1: unknown name `unknown_value`.

Program stopped with error.

Logs should not leak sensitive data by default. If a program handles private information, the Runtime should avoid recording full values unless debug mode explicitly allows it.

13.12 Registered action implementation

Registered actions are capabilities known to the Runtime.

They may be implemented in Python or another backend language, but they must be exposed to INTENTO through a controlled interface.

A registered action should declare its name, namespace, accepted parameter types, return type, safety level, description, and implementation function.

A conceptual action definition may look like this:

name: text.slugify

accepts: text

returns: text

safety: safe

description: Converts text into a lowercase URL-safe slug.

backend: python:functions.text.slugify

An INTENTO program can then call it:

use library "text"

remember title as "Human Readable Code"

use action "text.slugify" with title and call the result slug

show slug

Output:

human-readable-code

The program does not see or execute raw Python. It only calls a registered action.

If an action is not registered, it is not available.

use action "text.hidden_function" with "Hello" and call the result value

Expected error:

Semantic error: unknown registered action `text.hidden_function`.

This protects the Runtime from becoming an uncontrolled bridge to backend code.

13.13 Python-backed execution rules

Python can be an excellent backend for INTENTO because it is mature, practical, and has a large ecosystem. But Python-backed power must remain controlled.

INTENTO source should not contain raw Python code.

Invalid:

use python code "print('hello')"

Expected error:

Safety error: raw Python code is not allowed in INTENTO programs.

The correct model is registration.

A Python function can be registered by the Runtime as an INTENTO action. The INTENTO program then calls the action by name.

Example:

use library "text"

use action "text.count_words" with "INTENTO is readable" and call the result count

show count

Output:

3

The Runtime may implement `text.count_words` in Python, but the INTENTO source remains clean.

A Python-backed action should not be able to escape its declared safety level. If an action is declared safe, it should not write files, access the network, or execute system commands internally. If it needs those capabilities, it must be declared with the correct safety level and require the appropriate permissions.

The safety metadata must describe the real behavior, not the friendly name.

13.14 Module loading rules

Modules allow one INTENTO file to use actions from another INTENTO file.

When the Runtime sees:

use module "formatting.intento"

it should locate the module inside the allowed workspace, parse it, check it, and make its exported actions available to the current program.

If the module contains only action definitions, loading it should produce no visible output.

Module file:

```
to show_title with title:
    show "== " plus title plus " =="
```

Main program:

```
use module "formatting.intento"

show_title with "INTENTO"
```

Output:

```
== INTENTO ==
```

In strict mode, imported modules should not run top-level executable instructions during import.

Invalid module content in strict mode:

```
show "Loaded module"

to greet:
    show "Hello"
```

Expected error when imported:

```
Module error: top-level executable instructions are not allowed in imported module.
```

This rule keeps imports predictable. A module should provide definitions; the main program should decide what runs.

13.15 Dependency resolution

Before execution, the Runtime should resolve project dependencies.

Dependencies include modules, libraries, registered actions, and permissions.

Example:

```
use module "reporting.intento"

use library "text"
```

use action "text.slugify" with "Hello World" and call the result slug

show slug

Output:

hello-world

Before execution, the Runtime should verify that `reporting.intento` exists, that the `text` library is available, and that `text.slugify` is registered.

If the library is missing:

Library error: unknown library `text`.

If the action is missing:

Semantic error: unknown registered action `text.slugify`.

If the module is missing:

Module error: module not found: reporting.intento.

Dependency checks should happen early. A program should not run halfway and then discover that a required module or action is unavailable if that dependency could have been checked before execution.

13.16 Configuration and environment

A Runtime may support configuration through project metadata, command-line options, or environment settings.

Configuration may define the workspace, enabled libraries, allowed permissions, logging level, debug mode, and registered action sources.

Example command:

intento run main.intento --workspace ./my_project --debug

Possible output:

Workspace: ./my_project

Debug mode enabled.

Program started.

The workspace option tells the Runtime where file operations are allowed. Debug mode tells the Runtime to show more diagnostic information.

If the workspace does not exist, the Runtime should report an error.

intento run main.intento --workspace ./missing_workspace

Expected error:

Runtime error: workspace not found: ./missing_workspace.

Configuration should make execution more explicit, not more mysterious. If a program behaves differently because of configuration, the Runtime should make that visible when requested.

13.17 Testing INTENTO programs

A complete implementation should make INTENTO programs testable.

Testing does not need to be part of the core syntax at first. It can be supported by the Runtime.

A simple test command may run a program and compare its output with expected output.

Example:

intento test hello.intento

Possible output:

Test passed: hello.intento

If the output differs:

Test failed: hello.intento

Expected: Hello

Got: Hallo

A project may include test files in a `tests` folder:

my_project/

  main.intento

  tests/

    hello.test

Testing rules may evolve later, but the implementation principle is important: a language becomes more reliable when programs can be checked automatically.

For INTENTO, tests are especially useful because they confirm that human-readable code still produces precise behavior.

13.18 Implementation requirements

A correct INTENTO implementation should follow the official language reference.

It should parse the syntax exactly. It should report clear errors. It should preserve type rules. It should enforce workspace boundaries. It should apply confirmation and permission rules. It should expose registered actions through metadata. It should prevent raw shell and raw Python execution inside INTENTO source.

An implementation should not add silent behavior that changes the meaning of programs.

For example, this should not be silently accepted as numeric addition:

show "10" plus 5

Expected behavior depends on the official type rules. If text joining is allowed when text is involved, the output may be:

105

But this should not be treated as:

15

unless explicit conversion has occurred.

The implementation must not guess that `"10"` should become `10`.

The same principle applies everywhere. The Runtime should be helpful in its diagnostics, but strict in its execution.

13.19 Complete runtime example

This example shows how a full INTENTO project may run.

Project structure:

demo_project/

    main.intento

    intento.project

Project metadata:

runtime: INTENTO 1.0

entry: main.intento

libraries: text, file

permissions: read project files, write project files with confirmation

Program file, `main.intento`:

```
use library "text"
use library "file"

show "Demo project"

ask "Project title?" and call the answer title

if title is empty:
    show "Project title is required"
    stop

use action "text.slugify" with title and call the result slug

remember folder as "output/" plus slug
remember readme as folder plus "/README.txt"
remember content as "# " plus title

with confirmation:
    create folder folder
    create file readme with content

show "Created project folder: " plus folder
```

Example command:

intento run demo_project

Example interaction:

Demo project

Project title?

INTENTO Runtime

Confirm operation: create folder output/intento-runtime?

Confirm operation: create file output/intento-runtime/README.txt?

Created project folder: output/intento-runtime

Expected filesystem result inside the project workspace:

output/intento-runtime/

output/intento-runtime/README.txt

Expected README content:

INTENTO Runtime

Possible Runtime log:

Program started: demo_project/main.intento

Loaded library: text

Loaded library: file

Called action: text.slugify

Confirmed: create folder output/intento-runtime

Confirmed: create file output/intento-runtime/README.txt

Program finished successfully.

This example brings the whole system together. The project declares its requirements. The Runtime loads the correct entry point. The program asks for input, validates it, uses a registered action, creates files inside the workspace, asks for confirmation, writes logs, and finishes clearly.

That is the intended execution model of INTENTO.

13.20 Summary

The INTENTO Runtime is the bridge between readable source code and real execution.

It reads `.intento` files, parses instructions, checks meaning and types, applies safety rules, loads modules and libraries, calls registered actions, manages memory and scope, produces output, and records logs.

The project format makes larger programs explicit. The `intento.project` file declares the entry point, Runtime version, required libraries, and permissions. The workspace defines where file operations are allowed. Permissions define what a program may do. Confirmation protects sensitive operations.

Python may power registered actions underneath, but INTENTO source must remain controlled. Raw Python and raw shell execution do not belong inside ordinary INTENTO programs.

The implementation rule is simple: the Runtime may be powerful, but it must not be vague.

A correct INTENTO implementation should always preserve four promises.

The source is readable.
The syntax is parseable.
The behavior is predictable.
The execution is safe.

With those rules, INTENTO is not only a language idea. It is an implementable system.

Appendix — Word Operators and Symbol Aliases

INTENTO is designed to be human-readable first.

For this reason, its canonical arithmetic operators are written as words:

plus

minus

times

divided by

These operators make INTENTO source code understandable even to someone who is not used to programming syntax. A line such as:

show price plus tax

can be read almost like ordinary English. This is one of the core ideas of the language: code should express intention clearly before it becomes technical execution.

However, real programs can grow. In longer expressions, symbolic operators may be more compact and easier to scan for experienced programmers. For this reason, INTENTO allows symbolic aliases for the canonical word operators.

The word operators remain the official form.
The symbols are compact aliases.

A.1 Canonical operators

The official INTENTO arithmetic operators are:

plus

minus

times

divided by

Their meanings are straightforward.

The operator `plus` adds numbers or joins text.

show 10 plus 5

show "Hello " plus "INTENTO"

Output:

15

Hello INTENTO

The operator `minus` subtracts numbers.

show 10 minus 5

Output:

5

The operator `times` multiplies numbers.

show 10 times 5

Output:

50

The operator `divided by` divides numbers.

show 10 divided by 5

Output:

2

These forms are the canonical forms of INTENTO. Documentation, tutorials, beginner examples, and official explanations should prefer them.

A.2 Symbol aliases

INTENTO may also accept the following symbolic aliases:

plus +

minus -

times *

divided by /

The symbolic form has the same meaning as the word form.

Example:

show 10 + 5

show 10 - 5

show 10 * 5

show 10 / 5

Output:

15

5

50

2

These symbols do not introduce a second arithmetic system. They are only shorter spellings of the same operators.

The following two instructions are equivalent:

show 10 plus 5

show 10 + 5

Output:

15

15

A Runtime that supports symbol aliases must treat them according to the same type rules, precedence rules, and error rules as their word equivalents.

A.3 Why both forms exist

The word form exists because INTENTO wants to preserve controlled human readability.

This is especially useful in simple programs, teaching material, automation scripts, and code written for people who are not professional programmers.

Example:

remember total as price plus tax

show total

This line is easy to read aloud and easy to explain.

The symbolic form exists because some expressions become easier to write and review when they are compact.

Example:

remember final_price as (price + tax) * discount

show final_price

The same expression may also be written with canonical word operators:

remember final_price as (price plus tax) times discount

show final_price

Both are valid if the Runtime supports symbolic aliases.

The recommended style is simple: use word operators when clarity is more important, and use symbol aliases when compactness makes the expression easier to read.

A.4 Precedence

Symbol aliases must follow the same precedence rules as word operators.

Multiplication and division are evaluated before addition and subtraction.

show 2 plus 3 times 4

show 2 + 3 * 4

Output:

14

14

Parentheses may be used to control evaluation order.

show (2 plus 3) times 4

show (2 + 3) * 4

Output:

20

20

This rule matters because word operators and symbol aliases should not behave differently. The choice between them should affect readability, not meaning.

A.5 Type rules

Symbol aliases must follow the same type rules as word operators.

The `plus` operator and the `+` alias may add numbers.

show 10 plus 5

show 10 + 5

Output:

15

15

They may also join text.

show "Hello " plus "Aurora"

show "Hello " + "Aurora"

Output:

Hello Aurora

Hello Aurora

The `minus`, `times`, and `divided by` operators, and their aliases, require numbers.

Invalid:

show "10" - 5

Expected error:

Type error: `-` requires numbers.

Equivalent invalid word form:

show "10" minus 5

Expected error:

Type error: `minus` requires numbers.

The Runtime should not silently convert text to number. If the programmer wants numeric behavior, conversion must be explicit.

remember value_text as "10"

convert value_text to number and call it value

show value - 5

Output:

5

Explicit conversion keeps INTENTO predictable.

A.6 Mixing word and symbol operators

A Runtime may allow word operators and symbol aliases in the same expression, but programmers should avoid unnecessary mixing.

Valid but not recommended:

show 10 plus 5 * 2

Output:

20

Better canonical form:

show 10 plus 5 times 2

Output:

20

Better compact form:

show 10 + 5 * 2

Output:

20

Mixing is not wrong if the Runtime supports it, but style should serve readability. In most cases, a program should either use the word form for clarity or the symbolic form for compact expressions.

A.7 Documentation style

Official beginner documentation should prefer word operators.

Example:

remember total as price plus tax

show total

Advanced documentation may introduce symbolic aliases once the canonical form is understood.

Example:

remember total as price + tax

show total

The symbolic form should never replace the word form as the identity of INTENTO. Symbols are useful, but they are not the reason the language exists.

The readable form is the language’s center of gravity.

A.8 Runtime compatibility

A minimal INTENTO Core Runtime may support only word operators.

A fuller Runtime may support both word operators and symbolic aliases.

If a Runtime does not support symbolic aliases, it should report a clear error.

show 10 + 5

Expected error in a word-only Runtime:

Syntax error: symbol operator `+` is not supported by this Runtime.

Expected output in a Runtime that supports aliases:

15

This allows INTENTO to remain simple in its smallest form while still growing toward more comfortable programming syntax in fuller implementations.

A project may declare that it requires symbol alias support.

Possible project metadata:

runtime: INTENTO 1.0

features: symbol operators

If the Runtime does not support the feature, it should fail before execution.

Expected error:

Project error: required feature `symbol operators` is not supported.

A.9 Final rule

The official INTENTO operator rule is:

Word operators are canonical.

Symbol operators are aliases.

Both must behave the same when supported.

The word form protects INTENTO’s human-readable identity.

The symbolic form helps larger programs stay compact.

Neither form should introduce hidden conversion, different precedence, or different safety behavior.

A program written with words should be clear.
A program written with symbols should be compact.
A correct Runtime should make both precise.

Final Conclusions

INTENTO began with a simple question: can a programming language be closer to human intention without losing the precision required by computation?

This manual answers yes, but with an important condition.

A human-readable language must not become a vague language.

The central achievement of INTENTO is the separation between readable expression and controlled execution. The programmer writes instructions that are clear to a human reader. The Runtime parses those instructions, checks their structure, verifies names and types, enforces safety rules, loads libraries and modules, calls registered actions, and reports errors clearly.

This separation allows INTENTO to remain simple on the surface while still being technically serious underneath.

The language core is intentionally small. It defines memory with `remember`, output with `show`, input with `ask`, decisions with `if` and `else`, repetition with `repeat` and `for each`, reusable behavior with `to` and `return`, explicit conversion with `convert`, recoverable error handling with `try` and `on error`, and controlled sensitive operations with `with confirmation`.

The Standard Library extends this core without overwhelming it. Text, numbers, lists, files, dates, JSON, CSV, logs, system actions, and network actions are exposed through registered namespaces. Each action must define what it accepts, what it returns, what it does, and what safety level it requires.

The Runtime completes the picture. INTENTO is not only syntax. It is a system of execution. Its files, project format, workspace rules, permissions, confirmations, logs, CLI behavior, module loading, and Python-backed action registry are all part of making the language practical.

This manual also makes a deliberate choice about power. INTENTO does not expose raw Python or raw shell execution inside ordinary source code. That kind of unrestricted access would make the language powerful, but it would also make it unsafe and unclear. Instead, INTENTO uses registered actions. The backend may be powerful, but the INTENTO layer remains readable and controlled.

The result is not a replacement for Python. Not yet, and perhaps not in the same way.

Python is a mature general-purpose language with a vast ecosystem. INTENTO is a language of intention: a readable layer that can rely on mature backends while offering a clearer surface for humans. Its strength is not in imitating Python syntax. Its strength is in asking what programming looks like when intention is treated as a first-class part of the language.

This document closes the INTENTO 0.1 language specification.

It defines the philosophy, the syntax, the execution model, the safety model, the Standard Library, the Runtime expectations, and the operator rules.

What comes next is implementation.

The next step is to build a working Runtime: a parser, an interpreter, a memory model, an error system, a filesystem sandbox, a confirmation engine, a registry for actions, a small Standard Library, and a command-line interface capable of running `.intento` files.

Only then will INTENTO move from specification to execution.

But the foundation is now clear.

INTENTO is not free natural language.

INTENTO is not hidden Python.

INTENTO is not shell scripting with friendlier words.

INTENTO is a human-readable programming language with controlled syntax and real execution.

Its promise is simple:

write intention clearly,
execute it safely,
extend it carefully.